

Copyright © 2006-2008

如果你在Java开发工作中遇到困难,或者学习遇到瓶颈，欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛，为您解决Java工作、学习中遇到的问题，欢迎菜鸟，老鸟入坑，进群后群管理会私聊主动分享资料：Redis/MongoDB/MySQL/Zookeeper/Spring Cloud/Dubbo/Kafka/Hadoop/Hbase/Flink等高并发分布式/大数据等，期待你的一起加入。

VISIT…

LANZAROTE
Caliente.com

版权 ... xi
1. Creative Commons BY-ND-NC ... xi
序: Beta 0.2 ... xiii
前言 ... xiv
1. 如何使用本书 ... xiv
2. 你的反馈 ... xv
3. 字体约定[3] ... xv
4. Maven书写约定[4] ... xv
5. 致谢 ... xvi
1. 介绍 Apache Maven ... 1
1.1. Maven... 它是什么? ... 1
1.2. 约定优于配置（Convention Over Configuration） ... 1
1.3. 一个一般的接口 ... 2
1.4. 基于Maven插件的全局性重用 ... 3
1.5. 一个“项目”的概念模型 ... 4
1.6. Maven是Ant的另一种选择么? ... 4
1.7. 比较Maven和Ant ... 5
1.8. 总结 ... 9
2. 安装和运行Maven ... 10
2.1. 验证你的Java安装 ... 10
2.2. 下载Maven ... 10
2.3. 安装Maven ... 10
2.3.1. 在Mac OSX上安装Maven ... 11
2.3.2. 在Microsoft Windows上安装Maven ... 11
2.3.3. 在Linux上安装Maven ... 12
2.3.4. 在FreeBSD或OpenBSD上安装Maven ... 12
2.4. 验证Maven安装 ... 12
2.5. Maven安装细节 ... 12
2.5.1. 用户相关配置和仓库 ... 13
2.5.2. 升级Maven ... 13
2.6. 获得Maven帮助 ... 13
2.7. 使用Maven Help插件 ... 14
2.7.1. 描述一个Maven插件 ... 15
2.8. 关于Apache软件许可证 ... 17
I. Maven实战 ... 19
3. 一个简单的Maven项目 ... 20
3.1. 简介 ... 20
3.1.1. 下载本章的例子 ... 20
3.2. 创建一个简单的项目 ... 20
3.3. 构建一个简单的项目 ... 22

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

3.4. 简单的项目对象模型 (Project Object Model) 23
3.5. 核心概念 24
3.5.1. Maven插件和目标 (Plugins and Goals) 25
3.5.2. Maven生命周期 (Lifecycle) 26
3.5.3. Maven坐标 (Coordinates) 32
3.5.4. Maven仓库(Repositories) 34
3.5.5. Maven依赖管理 (Dependency Management) 36
3.5.6. 站点生成和报告 (Site Generation and Reporting) 38
3.6. 小结 38
4. 定制一个Maven项目 39
4.1. 介绍 39
4.1.1. 下载本章样例 39
4.2. 定义Simple Weather项目 39
4.2.1. Yahoo! Weather RSS 39
4.3. 创建Simple Weather项目 40
4.4. 定制项目信息 41
4.5. 添加新的依赖 43
4.6. Simple Weather源码 45
4.7. 添加资源 51
4.8. 运行Simple Weather项目 52
4.8.1. Maven Exec 插件 53
4.8.2. 浏览你的项目依赖 53
4.9. 编写单元测试 55
4.10. 添加测试范围依赖 58
4.11. 添加单元测试资源 58
4.12. 执行单元测试 61
4.12.1. 忽略测试失败 62
4.12.2. 跳过单元测试 63
4.13. 构建一个打包好的命令行应用程序 63
5. 一个简单的Web应用 66
5.1. 介绍 66
5.1.1. 下载本章样例 66
5.2. 定义这个简单的Web应用 66
5.3. 创建这个简单的Web应用 66
5.4. 配置Jetty插件 68
5.5. 添加一个简单的Servlet 70
5.6. 添加J2EE依赖 72
5.7. 小结 74
6. 一个多模块项目 75
6.1. 简介 75

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

6.1.1. 下载本章样例 75
6.2. simple-parent 项目 75
6.3. simple-weather 模块 77
6.4. simple-webapp 模块 80
6.5. 构建这个多模块项目 82
6.6. 运行Web应用 84
7. 多模块企业级项目 85
7.1. 简介 85
7.1.1. 下载本章样例 85
7.1.2. 多模块企业级项目 85
7.1.3. 本例中所用的技术 87
7.2. simple-parent项目 88
7.3. simple-model模块 90
7.4. simple-weather模块 94
7.5. simple-persist模块 99
7.6. simple-webapp模块 108
7.7. 运行这个Web应用 118
7.8. simple-command模块 120
7.9. 运行这个命令行程序 126
7.10. 小结 129
7.10.1. 编写接口项目程序 129
8. 优化和重构POM 131
8.1. 简介 131
8.2. POM清理 131
8.3. 优化依赖 132
8.4. 优化插件 136
8.5. 使用Maven Dependency插件进行优化 137
8.6. 最终的POM 140
8.7. 小结 150
II. Maven参考 151
9. 项目对象模型 152
9.1. 简介 152
9.2. POM 152
9.2.1. 超级POM 154
9.2.2. 最简单的POM 156
9.2.3. 有效POM 157
9.2.4. 真正的POM 157
9.3. POM语法 158
9.3.1. 项目版本 158
9.3.2. 属性引用 160

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

9.4. 项目依赖 161
9.4.1. 依赖范围 162
9.4.2. 可选依赖 163
9.4.3. 依赖版本界限 165
9.4.4. 传递性依赖 166
9.4.5. 冲突解决 167
9.4.6. 依赖管理 169
9.5. 项目关系 171
9.5.1. 坐标详解 171
9.5.2. 多模块项目 172
9.5.3. 项目继承 174
9.6. POM最佳实践 176
9.6.1. 依赖归类 176
9.6.2. 多模块 vs. 继承 178
10. 构建生命周期 184
10.1. 简介 184
10.1.1. 清理生命周期 (clean) 184
10.1.2. 默认生命周期 (default) 188
10.1.3. 站点生命周期 (site) 190
10.2. 打包相关生命周期 190
10.2.1. JAR 190
10.2.2. POM 191
10.2.3. Maven Plugin 191
10.2.4. EJB 192
10.2.5. WAR 193
10.2.6. EAR 193
10.2.7. 其它打包类型 194
10.3. 通用生命周期目标 195
10.3.1. Process Resources 195
10.3.2. Compile 199
10.3.3. Process Test Resources 200
10.3.4. Test Compile 200
10.3.5. Test 201
10.3.6. Install 202
10.3.7. Deploy 202
11. 构建Profile 203
11.1. Profile是用来做什么的? 203
11.1.1. 什么是构建可移植性 203
11.1.2. 选择一个适当级别的可移植性 204
11.2. 通过Maven Profiles实现可移植性 205

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

11.2.1. 覆盖一个项目对象模型 207
11.3. 激活Profile 208
11.3.1. 激活配置 210
11.3.2. 通过属性缺失激活 212
11.4. 外部Profile 212
11.5. Settings Profile 213
11.5.1. 全局Settings Profile 215
11.6. 列出活动的Profile 215
11.7. 提示和技巧 216
11.7.1. 常见的环境 216
11.7.2. 安全保护 218
11.7.3. 平台分类器 219
11.8. 小结 222
12. Maven套件 223
12.1. 简介 223
12.2. Assembly基础 223
12.2.1. 预定义的套件描述符 224
12.2.2. 构建一个套件Building an Assembly 225
12.2.3. 套件作为依赖 227
12.2.4. 通过套件依赖组装套件 228
12.3. 套件描述符概述 232
12.4. 套件描述符 234
12.4.1. 套件描述符中的属性引用 234
12.4.2. 必须的套件信息 234
12.5. 控制一个套件的内容 236
12.5.1. Files 元素 236
12.5.2. FileSets 元素 237
12.5.3. fileSets####### 239
12.5.4. dependencySets 元素 241
12.5.5. moduleSets 元素 250
12.5.6. Repositories元素 256
12.5.7. 管理套件的根目录 256
12.5.8. componentDescriptors和containerDescriptorHandlers ... 257
12.6. 最佳实践 258
12.6.1. 标准的，可重用的套件描述符 258
12.6.2. 分发（聚合）套件 261
12.7. 总结 265
13. 属性和资源过滤 266
13.1. 简介 266
13.2. Maven属性 266

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

13.2.1. Maven项目的属性 267
13.2.2. Maven的Settings属性 268
13.2.3. 环境变量属性 269
13.2.4. Java系统属性 269
13.2.5. 用户定义的属性 271
13.3. 资源过滤 272
14. Maven和Eclipse: m2eclipse 276
14.1. 简介 276
14.2. m2eclipse 276
14.3. 安装 m2eclipse 插件 277
14.3.1. 安装前提条件 277
14.3.2. 安装 m2eclipse 279
14.4. 开启 Maven 控制台 279
14.5. 创建一个 Maven 项目 280
14.5.1. 从 SCM 签出一个 Maven 项目 281
14.5.2. 用Maven Archetype创建一个Maven项目 283
14.5.3. 创建一个 Maven 模块 285
14.6. 创建一个Maven POM文件 287
14.7. 导入Maven项目 290
14.7.1. 导入一个Maven项目 292
14.7.2. 具体化一个Maven项目 293
14.8. 运行Maven构建 296
14.9. 使用Maven进行工作 298
14.9.1. 添加及更新依赖或插件 300
14.9.2. 创建一个Maven模块 301
14.9.3. 下载源码 302
14.9.4. 打开项目页面 302
14.9.5. 解析依赖 302
14.10. 使用Maven仓库进行工作 302
14.10.1. 搜索 Maven 构件和 Java 类 303
14.10.2. 为Maven仓库编制索引 306
14.11. 使用基于表单的POM编辑器 308
14.12. 在m2eclipse中分析项目依赖 317
14.13. Maven 选项 322
14.14. 小结 327
15. 站点生成 328
15.1. 简介 328
15.2. 使用Maven构建项目站点 328
15.3. 自定义站点描述符 331
15.3.1. 自定义页面顶端图片 332

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

15.3.2. 自定义导航菜单 333
15.4. 站点目录结构 334
15.5. 编写项目文档 335
15.5.1. APT样例 336
15.5.2. FML样例 336
15.6. 部署你的项目web站点 337
15.6.1. 配置服务器认证 338
15.6.2. 配置文件和目录模式 339
15.7. 自定义站点外观 339
15.7.1. 自定义站点CSS 339
15.7.2. 创建自定义的站点模板 340
15.7.3. 可重用的web站点皮肤 345
15.7.4. 创建自定义的主题CSS 346
15.7.5. 在皮肤中自定义站点模板 347
15.8. 提示与技巧 348
15.8.1. 给HEAD嵌入XHTML 348
15.8.2. 在你站点logo下添加链接 349
15.8.3. 为你的站点添加导航链接 349
15.8.4. 添加项目版本 350
15.8.5. 修改发布日期格式和位置 351
15.8.6. 使用Doxia宏 352
16. 仓库管理器 354
16.1. 简介 354
16.1.1. Nexus历史 354
16.2. 安装Nexus 355
16.2.1. 从Sonatype下载Nexus 355
16.2.2. 安装Nexus 355
16.2.3. 运行Nexus 355
16.2.4. 安装后检查单 357
16.2.5. 为Redhat/Fedora/CentOS设置启动脚本 358
16.2.6. 升级Nexus版本 360
16.3. 使用Nexus 361
16.3.1. 浏览仓库 362
16.3.2. 浏览组 364
16.3.3. 搜索构件 366
16.3.4. 浏览系统RSS源 367
16.3.5. 浏览日志文件和配置 369
16.3.6. 更改你的密码 370
16.4. 配置Maven使用Nexus 371
16.4.1. 使用Nexus中央代理仓库 371

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

16.4.2. 使用Nexus作为快照仓库 372
16.4.3. 为缺少的依赖添加仓库 374
16.4.4. 添加一个新的仓库 375
16.4.5. 添加一个仓库至一个组 377
16.5. 配置Nexus 379
16.5.. 定制服务器配置 379
16.5.2. 管理仓库 381
16.5.3. 管理组 386
16.5.4. 管理路由 388
16.5.5. 网络配置 391
16.6. 维护仓库 391
16.7. 部署构件至Nexus 392
16.7.1. 部署发布版 393
16.7.2. 部署快照版 394
16.7.3. 部署第三方构件 395
17. 编写插件 397
17.1. 简介 397
17.2. Maven编程 397
17.2.1. 什么是反转控制？ 397
17.2.2. Plexus简介 398
17.2.3. 为什么使用Plexus? 399
17.2.4. 什么是插件？ 399
17.3. 插件描述符 400
17.3.1. 顶层插件描述符元素 402
17.3.2. Mojo配置 402
17.3.3. 插件依赖 405
17.4. 编写自定义插件 405
17.4.1. 创建一个插件项目 405
17.4.2. 一个简单的Java Mojo 406
17.4.3. 配置插件前缀 408
17.4.4. 插件中的日志 411
17.4.5. Mojo类注解 412
17.4.6. 当Mojo失败的时候 414
17.5. Mojo参数 415
17.5.1. 为Mojo参数提供值 415
17.5.2. 多值的Mojo参数 417
17.5.3. 依赖于一个Plexus组件 419
17.5.4. Mojo参数注解 419
17.6. 插件和Maven生命周期 420
17.6.1. 执行平行的生命周期 421

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

17.6.2. 创建自定义的生命周期 421
17.6.3. 覆盖默认生命周期 423
18. 使用可选语言编写插件 425
18.1. 使用Ant编写插件 425
18.2. 创建一个Ant插件 425
18.3. 使用JRuby编写插件 428
18.3.1. 创建一个JRuby插件 429
18.3.2. Ruby Mojo实现 431
18.3.3. Ruby Mojo中使用日志 434
18.3.4. Raise一个MojoError 434
18.3.5. 在JRuby中引用Plexus组件 435
18.4. 使用Groovy编写插件 436
18.4.1. 创建一个Groovy插件 436
A. 附录: Settings细节 439
A.1. 简介 439
A.2. Settings细节 439
A.2.1. 简单值 439
A.2.2. 服务器（Servers） 441
A.2.3. 镜像（Mirrors） 442
A.2.4. 代理（Proxies） 443
A.2.5. Profiles 444
A.2.6. 激活（Activation） 444
A.2.7. 属性（Properties） 446
A.2.8. 仓库（Repositories） 447
A.2.9. 插件仓库 449
A.2.10. 激活的Profile 449
B. 附录: Sun规格说明可选实现 450

版权

Copyright © 2008 Sonatype, Inc.

在线版本由Sonatype, Inc.发布, 654 High Street, Suite 220, Palo Alto, CA, 94301.

印刷版本由O’Reilly Media, Inc.发布, 1005 Gravenstein Highway North, Sebastopol, CA 95472.

Nutshell Handbook, the Nutshell Handbook logo, and the O’Reilly logo是O’Reilly Media, Inc.注册的商标. The Developer’s Notebook系列命名, 实验室手册视觉外观，以及相关的商业外观是O’Reilly Media, Inc.的商标。

Java™和所有基于Java的商标及logo都是Sun Microsystems, Inc.的商标或者在美利坚合众国和其它国家的注册商标。很多制造商和销售商用来区分其产品的命名都称之为商标。本书中这些命名出现的地方，以及Sonatype, Inc.，作为商标声明，所有这些命名都以大写的方式或者首字母大写的方式打印。

虽然在准备本书的过程中采取了很多预防措施，但是，如果由于使用本书包含信息导致破坏性后果，出版商和作者不对错误和疏忽承担责任。

1. Creative Commons BY-ND-NC

本作品使用Creative Commons Attribution-Noncommercial-No Derivative Works 3.0 United States许可证。要了解更多关于该许可证的信息，请访问http://creativecommons.org/licenses/by-nc-nd/3.0/us/。你可以在遵循以下条件的前提下，自由的共享，复制，分发，显示，以及传递该作品：

- 你必须使用链接http://www.sonatype.com注明这是Sonatype, Inc.的作品。
- 你不可以将该作品用于商业用途。
- 你不可以更改，转换该作品，不可以基于该作品进一步创作。

如果你要在web页面上再次分发该作品，你必须包含如下的链接，并且在将关于著作人的URL单独列成一行（移除所有反斜杠然后连接所有URL参数）：

```
<div xmlns:cc="http://creativecommons.org/ns#"
     about="http://creativecommons.org/license/results-one?q_1=2&q_1=1\
            &field_commercial=n&field_derivatives=n&field_jurisdiction=us\
            &field_format=StillImage&field_worktitle=Maven%3A+The+Definitive+Guide\
            &field_attribute_to_name=Sonatype%2C+Inc.\
            &field_attribute_to_url=http%3A%2F%2Fwww.sonatype.com\
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛


```
            &field_sourceurl=http%3A%2F%2Fwww.sonatype.com%2Fbook&field_morepermiss
            &lang=en_US&language=en_US&n_questions=3">
   <a rel="cc:attributionURL" property="cc:attributionName"
      href="http://www.sonatype.com">Sonatype, Inc.</a> /
   <a rel="license" href="http://creativecommons.org/licenses/by-nc-nd/3.0/us/">
     CC BY-NC-ND 3.0</a>
</div>
```

当在美利坚合众国之外的范围内下载到或者分发到该作品是，该作品需要由合适的Creative Commons Attribution-Noncommercial-No Derivative Works 3.0许可证移植版本保护。如果在该辖区内没有可用的Creative Commons Attribution-Noncommercial-No Derivative Works 3.0许可证，该作品应该受提供下载或者分发所在辖区的Creative Commons Attribution-Noncommercial-No Derivative Works 3.0许可证移植版本保护。关于Creative Commons可用的辖区完整列表可以在Creative Commons的国际站点找到：http://creativecommons.org/international。

如果在某个特定的辖区没有Creative Commons许可证的移植版本，该作品应当受通用的，未移植的Creative Commons Attribution-Noncommercial-No Derivative Works version 3.0许可证保护，见http://creativecommons.org/licenses/by-nc-nd/3.0/。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

序: Beta 0.2

我们现在处于Beta发布阶段。也许你会问这代表了什么，其实这只是说从现在开始到正式1.0版本发布，本书结构不会又太大的变化了。以后可能会有新的章节加入，也可能不会有。目前，我们关注于当前内容的清晰和完整，同时我们也会注意限制本书的大小和范围。

本书中你会看到Nexus和m2eclipse两章拥有大量的内容和快照。为它们花大量气力是因为它们不仅仅是本书的某一章，同时也是自身项目的完整文档。

我们已经获得了关于本书的大量反馈，请不要停止给我们意见和建议，我们将非常感激，请发送至book@sonatype.com[1]。要了解本书的最近更新，关于英文版你可以关注该博客：http://blogs.sonatype.com/book；关于中文版你可以关注该博客：http://juvenshun.javaeye.com。

本书是开源的，这意味着你可以直接浏览该书的源码，并贡献你的一份力量：

- 中文版源码地址：http://github.com/sonatype/maven-guide-zh/tree/
- 英文版源码地址：http://github.com/sonatype/maven-guide-en/tree/

所有Sonatype的工作人员都为本书做了贡献，因此官方作者是"Sonatype"。

本书中文版由Juven Xu[2]翻译。

Tim O'Brien (tobrien@sonatype.com)

Evanston, IL

2008年9月1日

Juven Xu (juven@sonatype.com)

中国，苏州

2009年7曰3日

PS：你已经购买本书的英文印刷版本[3]了。

---

[1] mailto:book@sonatype.com
[2] http://juvenshun.javaeye.com
[3] http://www.amazon.com/Maven-Definitive-Guide-Sonatype-Company/dp/0596517335/ref=sr_1_1?ie=UTF8&s=books&qid=1223866448&sr=8-1

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

前言

Maven是一种构建工具，一种项目管理工具，一种用来运行构建任务的抽象容器。对于那些成熟的，希望用一致的方式来管理和构建大量相互依赖的模块和类库，并且使用了数千第三方组件的项目来说，Maven已经证明了它是一个不可缺少的工具。它帮助数百万的工程师从日常工作中减轻维护第三方依赖的负担，它帮助很多组织从构建管理的泥潭中挣脱出来，步入新的台阶，构建和维护软件所需要的工作不再是限制软件设计的因素。

本书首次尝试来全面解释Maven这一主题。它结合了所有之前Maven书籍作者的经验和工作，并且这还不是最终的结果，目前只是第一个版本而已，以后还会有更多的更新。虽然Maven已经出现有很多年了，但本书的作者们相信它还仅仅是刚开始兑现自己做的大胆的承诺。所有的作者，以及本书后面的公司，Sonatype[1]，相信本书的出版标志着围绕Maven变革和开发，以及其周围软件生态系统的一个新的阶段的开始。

1. 如何使用本书

拿起本书，随便翻阅浏览一下。读完了一页之后，如果你在阅读HTML版本，你可以点击链接到下一页，或者，如果你有纸质书，你会往下翻。如果你在电脑旁，你可以输入一些例子，跟随作者深入理解概念。但是，请不要在你生气的时候将这么大的一本书砸向任何一个人。

本书分成了三个部分：介绍性内容，第 I 部分 “Maven实战”和第 II 部分 “Maven参考”。介绍性内容包括了两章：第 1 章介绍 Apache Maven和第 2 章安装和运行Maven。第 I 部分 “Maven实战”开发一些实际的例子，并一步步分析这些例子的结构，同时循循善诱，详细解释，以此来解释Maven。如果你是Maven新手，应该从第 I 部分 “Maven实战”开始阅读。第 II 部分 “Maven参考”更多的是参考而非介绍，第 II 部分 “Maven参考”中的每一章集中讨论一个主题，并且会尽可能的深入每个主题的细节。例如，第 II 部分 “Maven参考”中的第 17 章编写插件一章通过一些例子和表格解释如何编写插件。

虽然第 I 部分 “Maven实战”和第 II 部分 “Maven参考”都提供了解释，但每个部分有不同的策略。第 I 部分 “Maven实战”更关注于Maven项目的上下文，而第 II 部分 “Maven参考”更关注于一个单独的主题。你可以跳跃着阅读本书，第 I 部分 “Maven实战”绝不是第 II 部分 “Maven参考”的前提，但是如果你在阅读第 II 部分 “Maven参考”之前阅读了第 I 部分 “Maven实战”，你会有更好的理解。学习Maven最好的方式是通过样例，但是当你完成这些样例之后，你就会需要一些优良的参考资料来帮助你在自己的环境中定制Maven。

[1] http://www.sonatype.com

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

2. 你的反馈

我们并非在写完本书之后，将一个Word文档扔给我们的出版商，然后举办一个Party来庆祝一项工作的完成。事实上，本书并没有“完成”；本书永远不会完全的“完成”。它所涉及的内容一直在变化，扩展，因此我们将本书的工作看成是一项与社区的持续会话。出版本书意味着真正的工作才开始，而你，我们的读者，扮演了一个很重要的角色，帮助我们完善本书。如果你看到本书的某些地方存在错误：拼写错误，代码错误，或者有明显的谎话，那么请告诉我们，给我们发邮件：book@sonatype.com[2]。

本书的进展依赖于你的反馈。我们想知道哪些地方是好的，哪些地方还不够好。我们想知道是否有无法理解的信息。如果你认为本书很糟糕，我们特别的想知道为什么。肯定和否定的意见都是欢迎的。当然，我们保留不认同的权利，但是所有反馈都会得到友好的回复。

3. 字体约定[3]

本书遵循了一些字体使用的约定。预先理解这些约定能让你更易使用本书。[3]

斜体
: 文件名，文件扩展名，URL，应用程序名，强调，以及一些首次介绍的新术语。

##
: Java类名，方法名，变量名，属性，数据类型，数据库元素，以及文字中的代码片段。

等宽粗体
: 命令行输入的命令，以及用来标亮插入到运行样例中的代码。

等宽斜体
: 用来标注输出。

4. Maven书写约定[4]

本书遵循一些与Apache Maven相关的命名和字体使用约定。同样，预先理解这些预定能让让你更易阅读本书。[4]

Compiler插件
: Maven插件首字母大写。

---

[2] mailto:tobrien@sonatype.com

[3]这些约定主要针对于英文版本，中文版可能有不适用的地方。

[4]这些约定主要针对于英文版本，中文版可能有不适用的地方。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

create目标
: Maven目标使用等宽字体。

"plugin"
: Maven大量使用了plug-in这一个概念，但是你会发现plugin没有在字典中定义。本书将该术语写成"plugin"是因为它更易书写和阅读，而且它是Maven社区中的一个标准。

Maven Lifecycle, Maven Standard Directory Layout, Maven Plugin, Project Object Model
: 不管在什么地方出现，核心的Maven概念都首字母大写。

goalParameter
: Maven目标参数使用等宽字体。

compile阶段
: 生命周期阶段使用等宽字体。

5. 致谢

Sonatype要感谢下面的贡献者。下面所列的人给了我们很多反馈，帮助我们提高了本书的质量。感谢Marcus Biel, Brian Dols, Mangalaganesh Balasubramanian, 以及Mark Stewart。特别感谢Joel Costigliola帮助我们调试和纠正Spring web一章。特别感谢来自Bamboo的Richard Coasby作为临时的语法咨询顾问。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

第 1 章 介绍 Apache Maven

虽然网络上有许多Maven的参考文章，但是没有一篇单独的，编写规范的介绍Maven的文字，它需要是一本细心编排的入门指南和参考手册。 我们做的，正是试图提供这样的，包含许多使用参考的文字。

1.1. Maven... 它是什么?

如何回答这个问题要看你怎么看这个问题。 绝大部分Maven用户都称Maven是一个"构建工具"：一个用来把源代码构建成可发布的构件的工具。 构建工程师和项目经理会说Maven是一个更复杂的东西：一个项目管理工具。那么区别是什么？ 像Ant这样的构建工具仅仅是关注预处理，编译，打包，测试和分发。像 Maven 这样的一个项目管理工具提供了构建工具所提供功能的超集。 除了提供构建的功能，Maven还可以生成报告，生成Web站点，并且帮助推动工作团 队成员间的交流。

一个更正式的 Apache Maven[1] 的定义： Maven是一个项目管理工具，它包含了一个项目对象模型 (Project Object Model)，一组标准集合，一个项目生命周期(Project Lifecycle)，一个依赖管理系统(Dependency Management System)，和用来运行定义在生命周期阶段(phase)中插件(plugin)目标(goal)的逻辑。 当你使用Maven的时候，你用一个明确定义的项目对象模型来描述你的项目，然后 Maven 可以应用横切的逻辑，这些逻辑来自一组共享的（或者自定义的）插件。

别让Maven是一个"项目管理"工具的事实吓跑你。如果你只是在找一个构建工具，Maven能做这个工作。 事实上，本书的一些章节将会涉及使用Maven来构建和分发你的项目。

1.2. 约定优于配置（Convention Over Configuration）

约定优于配置是一个简单的概念。 系统，类库，框架应该假定合理的默认值，而非要求提供不必要的配置。 流行的框架如 Ruby on Rails[2] 和 EJB3 已经开始坚持这些原则，以对像原始的 EJB 2.1 规范那样的框架的配置复杂度做出反应。 一个约定优于配置的例子就像 EJB3 持久化，将一个 特殊的Bean持久化，你所需要做的只是将这个类标注为 `@Entity` 。 框架将会假定表名和列名是基于类名和属性名。 系统也提供了一些钩子，当有需要的时候你可以重写这些名字，但是，在大部分情况下，你会发现使用框架提供的默认值会让你的项目运行的更快。

Maven通过给项目提供明智的默认行为来融合这个概念。 在没有自定义的情况下，源代码假定是在 `/usr/local/hudson/hudson-home/jobs/maven-guide-zh-to-production/workspace/content-zh/src/main/java`，资源文件假定是在 `/usr/local/hudson/hudson-`

[1] http://maven.apache.org
[2] http://www.rubyonrails.org/

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

home/jobs/maven-guide-zh-to-production/workspace/content-zh/src/main/resources 。测试代码假定是在 /usr/local/hudson/hudson-home/jobs/maven-guide-zh-to-production/workspace/content-zh/src/test 。项目假定会产生一个 JAR 文件。Maven 假定你想要把编译好的字节码放到 /usr/local/hudson/hudson-home/jobs/maven-guide-zh-to-production/workspace/content-zh/target/classes 并且在 /usr/local/hudson/hudson-home/jobs/maven-guide-zh-to-production/workspace/content-zh/target 创建一个可分发的 JAR 文件。 虽然这看起来无关紧要，但是想想大部分基于 Ant 的构建必须为每个子项目定义这些目录。 Maven 对约定优于配置的应用不仅仅是简单的目录位置，Maven 的核心插件使用了一组通用的约定，以用来编译源代码，打包可分发的构件，生成 web 站点，还有许多其他的过程。 Maven 的力量来自它的"武断"，它有一个定义好的生命周期和一组知道如何构建和装配软件的通用插件。如果你遵循这些约定，Maven 只需要几乎为零的工作——仅仅是将你的源代码放到正确的目录，Maven 将会帮你处理剩下的事情。

使用“遵循约定优于配置”系统的一个副作用是用户可能会觉得他们被强迫使用一种特殊的方法。 当然 Maven 有一些核心观点不应该被怀疑，但是其实很多默认行为还是可配置的。 例如项目源码的资源文件的位置可以被自定义，JAR 文件的名字可以被自定义，在开发自定义插件的时候，几乎任何行为可以被裁剪以满足你特定的环境需求。 如果你不想遵循约定，Maven 也会允许你自定义默认值来适应你的需求。

1.3. 一个一般的接口

在 Maven 为构建软件提供一个一般的接口之前，每个单独的项目都专门有人来管理一个完全自定义的构建系统。开发人员必须在开发软件之外去学习每个他们要参与的新项目的构建系统的特点。 在2001年，构建一个项目如 Turbine[3] 和构建另外一个项目如 Tomcat[4] ，两者方法是完全不同的。如果一个新的进行静态源码分析的源码分析工具面世了，或者如果有人开发了一个新的单元测试框架，每个人都必须放下手头的工作去想办法使这个新东西适应每个项目的自定义构建环境。 如何运行单元测试？世界上有一千种不同的答案。构建环境由无数无休止的关于工具和构建程序的争论所描述刻画。Maven 之前的时代是低效率的时代，是“构建工程师”的时代。

现在，大部分开源开发者已经或者正在使用 Maven 来管理他们新的软件项目。这种转变不仅仅是开发人员从一种构建工具转移到另外一种构建工具，更是开发人员开始为他们的项目采用一种一般的接口。随着软件系统变得越来越模块化，构建系统变得更复杂，而项目的数量更是如火箭般飞速上升。在 Maven 之前，当你想要从 Subversion 签出一个项目如 Apache ActiveMQ[5] 或 Apache ServiceMix[6] ，然后从源码进行构建，你需要为每个项目留出一个小时来理解给它的构建系统。这个项目需要构建什么？需要

[3] http://turbine.apache.org/
[4] http://tomcat.apache.org
[5] http://activemq.apache.org
[6] http://servicemix.apache.org

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

现在什么类库？把类库放哪里？构建中我该运行什么目标？最好的情况下，理解一个新项目的构建需要几分钟，最坏的情况下（例如 Jakarta 项目的旧的 Servlet API 实现），一个项目的构建特别的困难，以至于花了几个小时以后，新的贡献者也只能编辑源码和编译项目。现在，你只要签出源码，然后运行： mvn install 。虽然 Maven 有很多优点，包括依赖管理和通过插件重用一般的构建逻辑，但它成功的最核心原因是它定义了构建软件的一般的接口。每当你看到一个使用 Maven 的项目如 Apache Wicket[7]，你就可以假设你能签出它的源码然后使用 mvn install 构建它，没什么好争论的。你知道点火开关在哪里，你知道油门在右边，刹车在左边。

1.4. 基于Maven插件的全局性重用

Maven 的核心其实不做什么实际的事情，除了解析一些 XML 文档，管理生命周期与插件之外，它什么也不懂。Maven 被设计成将主要的职责委派给一组 Maven 插件，这些插件可以影响 Maven 生命周期，提供对目标的访问。绝大多数 Maven 的动作发生于 Maven 插件的目标，如编译源码，打包二进制代码，发布站点和其它构建任务。你从 Apache 下载的 Maven 不知道如何打包 WAR 文件，也不知道如何运行单元测试，Maven 大部分的智能是由插件实现的，而插件从 Maven 仓库获得。事实上，第一次你用全新的 Maven 安装运行诸如 mvn install 命令的时候，它会从中央 Maven 仓库下载大部分核心 Maven 插件。这不仅仅是一个最小化 Maven 分发包大小的技巧，这种方式更能让你升级插件以给你项目的构建提高能力。Maven 从远程仓库获取依赖和插件的这一事实允许了构建逻辑的全局性重用。

Maven Surefire 插件是负责运行单元测试的插件。从版本 1.0 发展到目前广泛使用的在 JUnit 基础上增加了 TestNG 测试框架支持的版本。这种发展并没有破坏向后兼容性，如果你使用之前 Surefire 插件编译运行你的 JUnit 3 单元测试，然后你升级到了最新版本的 Surefire 插件，你的测试仍然能成功运行。但是，我们又获得了新的功能，如果你想使用 TestNG 运行单元测试，那么感谢 Surefire 插件的维护者，你已经可以使用 TestNG 了。你也能运行支持注解的 JUnit 4 单元测试。不用升级 Maven 安装或者新装任何软件，你就能获得这些功能。更重要的是，除了 POM 中一个插件的版本号，你不需要更改你项目的任何东西。

这种机制不仅仅适用于 Surefire 插件，项目使用 Compiler 插件进行编译，通过 Jar 插件变成 JAR 文件，还有一些插件生成报告，运行 JRuby 和 Groovy 的代码，以及一些用来向远程服务器发布站点的插件。Maven 将一般的构建任务抽象成插件，同时这些插件得到了很好的维护以及全局的共享，你不需要从头开始自定义你项目的构建系统然后提供支持。你完全可以从 Maven 插件获益，这些插件有人维护，可以从远程仓库下载到。这就是基于 Maven 插件的全局性重用。

[7] http://wicket.apache.org

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

1.5. 一个“项目”的概念模型

Maven 维护了一个项目的模型，你不仅仅需要把源码编译成字节码，你还需要开发软件项目的描述信息，为项目指定一组唯一的坐标。你要描述项目的的属性。项目的许可证是什么？谁开发这个项目，为这个项目做贡献？这个项目依赖于其它什么项目没有？Maven不仅仅是一个“构建工具”，它不仅仅是在类似于 make 和 Ant 的工具的基础上的改进，它是包含了一组关于软件项目和软件开发的语义规则的平台。这个基于每一个项目定义的模型实现了如下特征：

依赖管理
: 由于项目是根据一个包含组标识符，构件标识符和版本的唯一的坐标定义的。项目间可以使用这些坐标来声明依赖。

远程仓库
: 和项目依赖相关的，我们可以使用定义在项目对象模型（POM）中的坐标来创建 Maven 构件的仓库。

全局性构建逻辑重用
: 插件被编写成和项目模型对象（POM）一起工作，它们没有被设计成操作某一个已知位置的特定文件。一切都被抽象到模型中，插件配置和自定义行为都在模型中进行。

工具可移植性/集成
: 像 Eclipse，NetBeans，和 InteliJ 这样的工具现在有共同的地方来找到项目的信息。在 Maven 出现之前，每个 IDE 都有不同的方法来存储实际上是自定义项目对象模型（POM）的信息。Maven 标准化了这种描述，而虽然每个 IDE 仍然继续维护它的自定义项目文件，但这些文件现在可以很容易的由模型生成。

便于搜索和过滤构件
: 像 Nexus 这样的工具允许你使用存储在 POM 中的信息对仓库中的内容进行索引和搜索。

Maven 为软件项目的语义一致性描述的开端提供了一个基础。

1.6. Maven是Ant的另一种选择么？

当然，Maven 是 Ant 的另一种选择，但是 Apache Ant[8] 继续是一个伟大的，被广泛使用的工具。它已经是多年以来 Java 构建的统治者，而你很容易的在你项目的 Maven 构建中集成 Ant 构建脚本。这是 Maven 项目一种很常见的使用模式。而另一方面，随着越来越多的开源项目转移到 Maven 用它作为项目管理平台，开发人员开始意识到

[8] http://ant.apache.org

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

Maven 不仅仅简化了构建管理任务，它也帮助鼓励开发人员的软件项目使用通用的接口。Maven 不仅仅是一个工具，它更是一个平台，当你只是将 Maven 考虑成 Ant 的另一种选择的时候，你是在比较苹果和橘子。“Maven”包含了很多构建工具以外的东西。

有一个核心观点使得所有的关于 Maven 和. Ant， Maven 和 Buildr， Maven 和 Grandle 的争论变得无关紧要。Maven并不是完全根据你构建系统的机制来定义的，它不是为你构建的不同任务编写脚本，它提倡一组标注，一个一般的接口，一个生命周期，一个标准的仓库格式，一个标准的目录布局，等等。它当然也不太在意 POM 的格式正好是 XML 还是 YAML 还是 Ruby。它比这些大得多，Maven 涉及的比构建工具本身多得多。当本书讨论 Maven 的时候，它也设计到支持 Maven 的软件，系统和标准。Buildr，Ivy，Gradle，所有这些工具都和 Maven 帮助创建的仓库格式交互，而你可以很容易的使用如 Nexus 这样的工具来支持一个完全由 Buildr 编写的构建。Nexus 将在本书后面介绍。

虽然 Maven 是很多类似工具的另一个选择？但社区需要向前发展，就要看清楚技术是资本经济中不友好的竞争者之间持续的、零和的游戏。这可能是大企业之前相互关联的方式，但是和开源社区的工作方式没太大关系。“谁是胜利者？Ant 还是 Maven”这个大标题没什么建设性意义。如果你非要我们来回答这个问题，我们会很明确的说作为构建的基本技术，Maven 是 Ant 的更好选择；同时，Maven 的边界在持续的移动，Maven 的社区也在持续的是试图找到新的方法，使其更通用，互操作性更好，更易协同工作。Maven 的核心财产是声明性构建，依赖管理，仓库管理，基于插件的高度和重用，但是当前，和开源社区相互协作以降低”企业级构建“的低效率这个目标来比，这些想法的特定实现没那么重要。

1.7. 比较Maven和Ant

虽然上一节应该已经让你确信本书的作者没有兴趣挑起 Apache Ant 和 Apache Maven 之间的争执，但是我们认识到许多组织必须在 Apache Ant 和 Apache Maven 之间做一个选择。本节我们对比一下这两个工具。

Ant 在构建过程方面十分优秀，它是一个基于任务和依赖的构建系统。每个任务包含一组由 XML 编码的指令。有 `copy` 任务和 `javac` 任务，以及 `jar` 任务。在你使用 Ant 的时候，你为 Ant 提供特定的指令以编译和打包你的输出。看下面的例子，一个简单的 `build.xml` 文件：

例 1.1. 一个简单的 Ant build.xml 文件

```
<project name="my-project" default="dist" basedir=".">
    <description>
        simple example build file
    </description>
  <!-- set global properties for this build -->
  <property name="src" location="src/main/java"/>
  <property name="build" location="target/classes"/>
  <property name="dist"  location="target"/>

  <target name="init">
    <!-- Create the time stamp -->
    <tstamp/>
    <!-- Create the build directory structure used by compile -->
    <mkdir dir="org.apache.maven.model.Build@d7e661"/>
  </target>

  <target name="compile" depends="init"
        description="compile the source " >
    <!-- Compile the java code from ${src} into org.apache.maven.model.Build@d7e661
    <javac srcdir="${src}" destdir="org.apache.maven.model.Build@d7e661"/>
  </target>

  <target name="dist" depends="compile"
        description="generate the distribution" >
    <!-- Create the distribution directory -->
    <mkdir dir="${dist}/lib"/>

    <!-- Put everything in org.apache.maven.model.Build@d7e661 into the MyProject-$
    <jar jarfile="${dist}/lib/MyProject-${DSTAMP}.jar" basedir="org.apache.maven.mo
  </target>

  <target name="clean"
        description="clean up" >
    <!-- Delete the org.apache.maven.model.Build@d7e661 and ${dist} directory trees
    <delete dir="org.apache.maven.model.Build@d7e661"/>
    <delete dir="${dist}"/>
  </target>
</project>
```

在这个简单的 Ant 例子中，你能看到，你需要明确的告诉 Ant 你想让它做什么。有一个包含 javac 任务的编译目标用来将 src/main/java 的源码编译至 target/classes 目录。你必须明确告诉 Ant 你的源码在哪里，结果字节码你想存储在哪里，如何将这

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

些字节码打包成 JAR 文件。虽然最近有些进展以帮助 Ant 减少程序，但一个开发者对 Ant 的感受是用 XML 编写程序语言。

用 Maven 样例与之前的 Ant 样例做个比较。在 Maven 中，要从 Java 源码创建一个 JAR 文件，你只需要创建一个简单的 pom.xml，将你的源码放在 /usr/local/hudson/hudson-home/jobs/maven-guide-zh-to-production/workspace/content-zh/src/main/java ，然后从命令行运行 mvn install。下面的样例 Maven pom.xml 文件能完成和之前 Ant 样例所做的同样的事情。

例 1.2. 一个简单的 Maven pom.xml

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.sonatype.mavenbook</groupId>
  <artifactId>my-project</artifactId>
  <version>1.0</version>
</project>
```

这就是你 pom.xml 的全部。从命令行运行 mvn install 会处理资源文件，编译源代码，运行单元测试，创建一个 JAR ，然后把这个 JAR 安装到本地仓库以为其它项目提供重用性。不用做任何修改，你可以运行 mvn site ，然后在 target/site 目录找到一个 index.html 文件，这个文件链接了 JavaDoc 和一些关于源代码的报告。

诚然，这是一个最简单的样例项目。一个只包含源代码并且生成一个 JAR 的项目。一个遵循 Maven 的约定，不需要任何依赖和定制的项目。如果我们想要定制行为，我们的 pom.xml 的大小将会增加，在最大的项目中，你能看到一个非常复杂的 Maven POM 的集合，它们包含了大量的插件定制和依赖声明。但是，虽然你项目的 POM 文件变得增大，它们包含的信息与一个差不多大小的基于 Ant 项目的构建文件的信息是完全不同的。Maven POM 包含声明：“这是一个 JAR 项目”，“源代码在 src/main/java 目录”。Ant 构建文件包含显式的指令：“这是一个项目”，“源代码在 src/main/java ”，“针对这个目录运行 javac ”，“把结果放到 target/classes ”，“从……创建一个 JAR ”，等等。Ant 必须的过程必须是显式的，而 Maven 有一些“内置”的东西使它知道源代码在哪里，如何处理它们。

该例中 Ant 和 Maven 的区别是：

Apache Ant

- Ant 没有正式的约定如一个一般项目的目录结构，你必须明确的告诉 Ant 哪里去找源代码，哪里放置输出。随着时间的推移，非正式的约定出现了，但是它们还没有在产品中模式化。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

- Ant 是程序化的，你必须明确的告诉 Ant 做什么，什么时候做。你必须告诉它去编译，然后复制，然后压缩。
- Ant 没有生命周期，你必须定义目标和目标之间的依赖。你必须手工为每个目标附上一个任务序列。

Apache Maven

- Maven 拥有约定，因为你遵循了约定，它已经知道你的源代码在哪里。它把字节码放到 `target/classes` ，然后在 `target` 生成一个 JAR 文件。
- Maven 是声明式的。你需要做的只是创建一个 pom.xml 文件然后将源代码放到默认的目录。Maven 会帮你处理其它的事情。
- Maven 有一个生命周期，当你运行 mvn install 的时候被调用。这条命令告诉 Maven 执行一系列的有序的步骤，直到到达你指定的生命周期。遍历生命周期旅途中的一个影响就是，Maven 运行了许多默认的插件目标，这些目标完成了像编译和创建一个 JAR 文件这样的工作。

Maven 以插件的形式为一些一般的项目任务提供了内置的智能。如果你想要编写运行单元测试，你需要做的只是编写测试然后放到 `/usr/local/hudson/hudson-home/jobs/maven-guide-zh-to-production/workspace/content-zh/src/test/java` ，添加一个对于 TestNG 或者 JUnit 的测试范围依赖，然后运行 mvn test 。如果你想要部署一个 web 应用而非 JAR ，你需要做的是改变你的项目类型为 war ，然后把你文档根目录置为 `/usr/local/hudson/hudson-home/jobs/maven-guide-zh-to-production/workspace/content-zh/src/main/webapp` 。当然，你可以用 Ant 做这些事情，但是你将需要从零开始写这些指令。使用 Ant ，你首先需要确定 JUnit JAR 文件应该放在哪里，然后你需要创建一个包含这个 JUnit JAR 文件的 classpath ，然后告诉 Ant 它应该从哪里去找测试源代码，编写一个目标来编译测试源代码为字节码，使用 JUnit 来执行单元测试。

没有诸如 antlibs 和 lvy 等技术的支持（即使有了这些支持技术），Ant 给人感觉是自定义的程序化构建。项目中一组高效的坚持约定的 Maven POM ，相对于 Ant 的配置文件，只有很少的 XML 。Maven 的另一个优点是它依靠广泛公用的 Maven 插件。所有人使用 Maven Surefire 插件来运行单元测试，如果有人添加了一些针对新的测试框架的支持，你可以仅仅通过在你项目的 POM 中升级某个特定插件的版本来获得新的功能。

使用 Maven 还是 Ant 的决定不是非此即彼的，Ant 在复杂的构建中还有它的位置。如果你目前的构建包含一些高度自定义的过程，或者你已经写了一些 Ant 脚本通过一种明确的方法完成一个明确的过程，而这种过程不适合 Maven 标准，你仍然可以在 Maven 中用这些脚本。作为一个 Maven 的核心插件， Ant 还是可用的。自定义的插件可以用 Ant 来实现，Maven 项目可以配置成在生命周期中运行 Ant 的脚本。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

1.8. 总结

我们刻意的使这篇介绍保持得简短。 我们略述了一些Maven定义的要点，它们合起来是什么，它是基于什么改进的，同时介绍了其它构建工具。 下一章将深入一个简单的项目，看 Maven 如何能够通过最小数量的配置来执行典型的任务。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

第 2 章 安装和运行Maven

本章包含了在许多不同平台上安装Maven的详细指令。我们会介绍得尽可能详细以减少安装过程中可能出现的问题，而不去假设你已经熟悉安装软件和设置环境变量。本章的唯一前提是你已经安装了恰当的JDK。如果你仅仅对安装感兴趣，读完下载Maven和安装Maven后，你就可以直接去看本书其它的部分。如果你对Maven安装的细节感兴趣，本章将会给你提供一个简要的介绍，以及Apache软件许可证，版本2.0的介绍。

2.1. 验证你的Java安装

尽管Maven可以运行在Java 1.4上，但本书假设你在至少Java 5上运行。尽管使用你操作系统上最新的稳定版本的JDK。本书的例子在Java 5或者Java 6上都能运行。

```
% java -version
java version "1.6.0_02"
Java(TM) SE Runtime Environment (build 1.6.0_02-b06)
Java HotSpot(TM) Client VM (build 1.6.0_02-b06, mixed mode, sharing)
```

Maven能在所有的验证过的Java™ 兼容的JDK上工作，也包括一些未被验证JDK实现。本书的所有样例是基于Sun官方的JDK编写和测试的。如果你正在使用Linux，你可能需要自己下载Sun的JDK，并且确定JDK的版本（运行 java -version）。目前Sun已经将Java开源，因此这种情况将来有希望得到改进，将来很有可能Sun JRE和JDK会被默认安装在Linux上。但直到现在为止，你还是需要自己去下载。

2.2. 下载Maven

你可以从Apache Maven项目的web站点下载Maven：http://maven.apache.org/download.html.

当你下载Maven的时候，确认你选择了最新版本的Apache Maven。本书书写的时候最新版本是Maven 2.0.9 。如果你不熟悉Apache软件许可证，你需要在使用产品之前去熟悉该许可证的条款。更多的关于Apache软件许可证的信息可以第 2.8 节 “关于Apache软件许可证”找到。

2.3. 安装Maven

操作系统之间有很大的区别，像Mac OSX 和微软的Windows，而且不同版本Windows之间也有微妙的差别。幸运的是，在所有操作系统上安装Maven的过程，相对来说还是比较直接的。下面的小节概括了在许多操作系统上安装Maven的最佳实践。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

2.3.1. 在Mac OSX上安装Maven

你可以从http://maven.apache.org/download.html下载Maven的二进制版本。下载最新的，下载格式最方便你使用的版本。找个地方存放它，并把存档文件解开。如果你把存档文件解压到 /usr/local/maven-2.0.9 ；你可能会需要创建一个符号链接，那样就能更容易使用，当你升级Maven的时候也不再需要改变环境变量。

```
/usr/local % ln -s maven-2.0.9 maven
/usr/local % export M2_HOME=/usr/local/maven
/usr/local % export PATH=/usr/local/maven/bin:/usr/local/bin:/usr/local/maven/bin:/
```

将Maven安装好后，你还需要做一些事情以确保它正确工作。你需要将它的 bin 目录（该例中为 /usr/local/maven/bin）添加到你的命令行路径下。你还需要设置 M2_HOME 环境变量，其对应值为Maven的根目录（该例中为 /usr/local/maven）。

注意

在OSX Tiger和OSX Leopard上安装指令是相同的。有报告称Maven 2.0.6正和XCode的预览版本一起发布。如果你安装了XCode，从命令行运行 mvn 检查一下它是否可用。 XCode把Maven安装在了/usr/share/maven。我们强烈建议安装最新版本的Maven 2.0.9，因为随着Maven 2.0.9的发布很多bug被修正了，还有很多改进。

你还需要把 M2_HOME 和 PATH 写到一个脚本里，每次登陆的时候运行这个脚本。把下面的几行加入到 .bash_login。

```
export M2_HOME=/usr/local/maven
export PATH=/usr/local/maven/bin:/usr/local/bin:/usr/local/maven/bin:/usr/kerberos/
```

一旦你把这几行加入到你的环境中，你就可以在命令行运行Maven了。

注意

这些安装指令假设你正运行bash。

2.3.2. 在Microsoft Windows上安装Maven

在Windows上安装Maven和在Mac OSX上安装Maven十分类似，最主要的区别在于安装位置和设置环境变量。在这里假设Maven安装目录是 c:\Program Files\maven-2.0.9 ，但是，只要你设置的正确的环境变量，把Maven安装到其它目录也一样。当你把Maven解压到安装目录后，你需要设置两个环境变量——PATH和M2_M2_HOME。设置这两个环境变量，键入下面的命令：

```
C:\Users\tobrien > set M2_HOME=c:\Program Files\maven-2.0.9
C:\Users\tobrien > set PATH=%PATH%;%M2_HOME%\bin
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

在命令行设置环境变量后，你可以在当前会话使用Maven，但是，除非你通过控制面板把它们加入系统变量，你将需要每次登陆系统的时候运行这两行命令。你应该在Microsoft Windows中通过控制面板修改这两个变量。

2.3.3. 在Linux上安装Maven

遵循第 2.3.1 节 “在Mac OSX上安装Maven”的步骤，在Linux机器上安装Maven。

2.3.4. 在FreeBSD或OpenBSD上安装Maven

遵循第 2.3.1 节 “在Mac OSX上安装Maven”的步骤，在FreeBSD或者OpenBSD机器上安装Maven。T

2.4. 验证Maven安装

当Maven安装完成后，你可以检查一下它是否真得装好了，通过在命令行运行 mvn -v。如果Maven装好了，你应该能看到类似于下面的输出。

```
$ mvn -v
Maven 2.0.9
```

如果你看到了这样的输出，那么Maven已经成功安装了。如果你看不到，而且你的操作系统找不到 mvn 命令，那么确认一下PATH和M2_HOME环境变量是否已经正确设置了。

2.5. Maven安装细节

Maven的下载文件只有大概1.5 MiB，它能达到如此苗条的大小是因为Maven的内核被设计成根据需要从远程仓库获取插件和依赖。当你开始使用Maven，它会开始下载插件到本地仓库中，就像第 2.5.1 节 “用户相关配置和仓库”所描述的那样。对此你可能比较好奇，让我们先很快的看一下Maven的安装目录是什么样子。

```
/usr/local/maven $ ls -p1
LICENSE.txt
NOTICE.txt
README.txt
bin/
boot/
conf/
lib/
```

LICENSE.txt 包含了Apache Maven的软件许可证。第 2.8 节 “关于Apache软件许可证”会详细描述该许可证。NOTICE.txt 包含了一些Maven依赖的类库所需要的通告及权限。README.txt包含了一些安装指令。 bin/目录包含了运行Maven的mvn脚本。 boot/ 目录包含了一个负责创建Maven运行所需要的类装载器的JAR文件

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

(classwords-1.1.jar)。 conf/ 目录包含了一个全局的settings.xml文件，该文件用来自定义你机器上Maven的一些行为。如果你需要自定义Maven，更通常的做法是覆写~/.m2目录下的settings.xml文件，每个用户都有对应的这个目录。lib/ 目录有了一个包含Maven核心的JAR文件(maven-2.0.9-uber.jar)。

2.5.1. 用户相关配置和仓库

当你不再仅仅满足于使用Maven，还想扩展它的时候，你会注意到Maven创建了一些本地的用户相关的文件，还有在你home目录的本地仓库。在~/.m2目录下有：

~/.m2/settings.xml
: 该文件包含了用户相关的认证，仓库和其它信息的配置，用来自定义Maven的行为。

~/.m2/repository/
: 该目录是你本地的仓库。当你从远程Maven仓库下载依赖的时候，Maven在你本地仓库存储了这个依赖的一个副本。

> **注意**
>
> 在Unix(和OSX)上，可以用 ~ 符号来表示你的home目录，（如~/bin表示/home/tobrien/bin）。在Windows上，我们仍然使用 ~ 来表示你的home目录。在Windows XP上，你的home目录是 C:\Documents and Settings\tobrien，在Windows Vista上，你的home目录是 C:\Users\tobrien。从现在开始，你应该能够理解这种路径表示，并翻译成你操作系统上的对应路径。

2.5.2. 升级Maven

如果你遵循第 2.3.1 节 “在Mac OSX上安装Maven”和第 2.3.3 节 “在Linux上安装Maven”，在Mac OSX或者Unix机器上安装了Maven。那么把Maven升级成较新的版本是很容易的事情。只要在当前版本Maven（/usr/local/maven-2.0.9）旁边安装新版本的Maven（/usr/local/maven-2.future），然后将符号链接 /usr/local/maven 从 /usr/local/maven-2.0.9 改成 /usr/local/maven-2.future即可。你已经将M2_HOME环境变量指向了 /usr/local/maven，因此你不需要更改任何环境变量。

如果你在Windows上安装了Maven，将Maven解压到 c:\Program Files\maven-2.future，然后更新你的M2_HOME环境变量。

2.6. 获得Maven帮助

虽然本书的目的是作为一本全面的参考手册，但是仍然会有一些主题我们会不小心遗漏，一些特殊情况和技巧我们也覆盖不到。Maven的核心十分简单，它所做的工作其实

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

都交给插件了。插件太多了，以至于不可能在一本书上全部覆盖。你将会碰到一些本书没有涉及的问题，碰到这种情况，我们建议你在下面的地址去寻找答案。

http://maven.apache.org
: 你首先应该看看这里，Maven的web站点包含了丰富的信息及文档。每个插件都有几页的文档，这里还有一系列“快速开始”的文档，它们是本书内容的十分有帮助的补充。虽然Maven站点包含了大量信息，它同时也可能让你迷惑沮丧。那里提供了一个自定义的Google搜索框，以用来搜索已有的Maven站点信息，它能比通用的Google搜索提供更好的结果。

Maven User Mailing List
: Maven用户邮件列表是用户问问题的地方。在你问问题之前，你可以先搜索一下之前的讨论，有可能找到相关问题。问一个已经问过的问题，而不先查一下该问题是否存在了，这种形式不太好。有很多有用的邮件列表归档浏览器，我们发现Nabble最有用。你可以在这里浏览邮件列表：http://www.nabble.com/Maven---Users-f178.html。你也可以按照这里的指令来加入用户邮件列表：http://maven.apache.org/mail-lists.html。

http://www.sonatype.com
: Sonatype维护了一个本书的在线副本，以及其它Maven相关的指南。

注意

除去一些专门的Maven贡献者所做的十分优秀的工作，Maven web站点组织的比较糟糕，有很多不完整的，甚至有时候有些误导人的文档片段。在整个Maven社区里，插件文档的一般标准十分缺乏，一些插件的文档十分的丰富，但是另外一些连基本的使用命令都没有。通常你最好是在用户邮件列表里面去搜索下解决方案。

2.7. 使用Maven Help插件

本书中，我们将会介绍Maven插件，讲述Maven项目对象模型（POM）文件，settings文件，还有profile。有些时候，你需要一个工具来帮助你理解一些Maven使用的模型，以及某个插件有什么可用的目标。Maven Help插件能让你列出活动的Maven Profile，显示一个实际POM（effective POM），打印实际settings（effective settings），或者列出Maven插件的属性。

注意

如果想看一下POM和插件的概念性介绍，参照第三章：一个简单的Maven项目。

Maven Help 插件有四个目标。前三个目标是—— active-profiles, effective-pom 和 effective-settings —— 描述一个特定的项目，它们必须在项目的目录下运行。 最后一个目标—— describe ——相对比较复杂，展示某个插件或者插件目标的相关信息。

help:active-profiles
: 列出当前构建中活动的Profile（项目的，用户的，全局的）。

help:effective-pom
: 显示当前构建的实际POM，包含活动的Profile。

help:effective-settings
: 打印出项目的实际settings, 包括从全局的settings和用户级别settings继承的配置。

help:describe
: 描述插件的属性。它不需要在项目目录下运行。但是你必须提供你想要描述插件的 groupId 和 artifactId。

2.7.1. 描述一个Maven插件

一旦你开始使用Maven，你会花很多时间去试图获得Maven插件的信息：插件如何工作？配置参数是什么？目标是什么？ 你会经常使用help:describe 目标来获取这些信息。通过 plugin 参数你可以指定你想要研究哪个插件，你可以传入插件的前缀（如 help 插件就是 maven-help-plugin），或者可以是 groupId:artifact[:version] ，这里 version 是可选的。比如， 下面的命令使用 help 插件的 describe 目标来输出 Maven Help 插件的信息。

```
$ mvn help:describe -Dplugin=help
...
Group Id:  org.apache.maven.plugins
Artifact Id: maven-help-plugin
Version:     2.0.1
Goal Prefix: help
Description:

The Maven Help plugin provides goals aimed at helping to make sense out of
    the build environment. It includes the ability to view the effective
    POM and settings files, after inheritance and active profiles
    have been applied, as well as a describe a particular plugin goal to give
    usage information.
...
```

通过设置 plugin 参数来运行 describe 目标，输出为该插件的Maven坐标，目标前缀，和该插件的一个简要介绍。尽管这些信息非常有帮助，你通常还是需要了解更多

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

的详情。如果你想要 Help 插件输出完整的带有参数的目标列表，只要运行带有参数 `full` 的 `help:describe` 目标就可以了，像这样：

```
$ mvn help:describe -Dplugin=help -Dfull
...
Group Id:  org.apache.maven.plugins
Artifact Id: maven-help-plugin
Version:     2.0.1
Goal Prefix: help
Description:

The Maven Help plugin provides goals aimed at helping to make sense out of
    the build environment. It includes the ability to view the effective
    POM and settings files, after inheritance and active profiles
    have been applied, as well as a describe a particular plugin goal to give usage
    information.

Mojos:

===============================================
Goal: 'active-profiles'
===============================================
Description:

Lists the profiles which are currently active for this build.

Implementation: org.apache.maven.plugins.help.ActiveProfilesMojo
Language: java

Parameters:
-----------------------------------------------

[0] Name: output
Type: java.io.File
Required: false
Directly editable: true
Description:

This is an optional parameter for a file destination for the output of this mojo...
listing of active profiles per project.

-----------------------------------------------

[1] Name: projects
Type: java.util.List
Required: true
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛


```
Directly editable: false
Description:

This is the list of projects currently slated to be built by Maven.

-----------------------------------------------

This mojo doesn't have any component requirements.
===============================================

... removed the other goals ...
```

该选项能让你查看插件所有的目标及相关参数。但是有时候这些信息显得太多了。这时候你可以获取单个目标的信息，设置 `mojo` 参数和 `plugin` 参数。下面的命令列出了 Compiler 插件的 `compile` 目标的所有信息

```
$ mvn help:describe -Dplugin=compiler -Dmojo=compile -Dfull
```

注意

什么？ Mojo ？在Maven里面， 一个插件目标也被认为是一个 “Mojo” 。

2.8. 关于Apache软件许可证

Apache Maven 是基于 Apache 许可证2.0版 发布的。如果你想阅读该许可证，你可以查阅 `/usr/local/maven/LICENSE.txt` 或者从开源发起组织的网站上查阅 http://www.opensource.org/licenses/apache2.0.php。

很有可能你正在阅读本书但你又不是律师。如果你想知道 Apache许可证2.0版意味着什么，Apache软件基金会收集了一个很有帮助的，关于许可证的常见问题解答（FAQ）：http://www.apache.org/foundation/licence-FAQ.html。这里是对问题“我不是律师，所有这些是什么意思？”的回答。

它允许你：

- • 自由的下载和使用 Apache 软件，无论是软件的整体还是部分，也无论是出于个人目的，公司内部目的，还是商业目的。

- • 在你创建的类库或分发版本里使用 Apache 软件。

它禁止你：

- 在没有正当的权限下重新分发任何源于 Apache 的软件或软件片段。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

- 以任何可能声明或暗示基金会认可你的分发版本的形式下使用 Apache 软件基金会拥有的标志。
- 以任何可能声明或暗示你创建了 Apache 软件的形式下使用 Apache 软件基金会拥有的标志。

它要求你：

- 在你重新分发的包含 Apache 软件的软件里，包含一份该许可证的副本。
- 对于任何包含 Apache 软件的分发版本，提供给 Apache软件基金会清楚的权限。

它不要求你：

- 在任何你再次发布的包含Apache软件的版本里，包含Apache软件本身源代码，或者你做的改动的源码。
- 提交你对软件的改动至 Apache 软件基金会（虽然我们鼓励这种反馈）。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

部分 I. Maven实战

第一本关于Maven的书是来自O'Reilly[1]的Maven开发者笔记[2]，这本书通过一系列步骤介绍Maven。开发者笔记系列书籍背后的想法是，当开发人员和另一个开发人员坐在一起，经历他曾经用来学习和编码的整个思考过程，这样会学得最好。虽然开发者笔记系列成功了，但笔记格式有一个缺点：笔记被设计成“集中关注于目标”，它让你通过一系列步骤来完成某个特定的目标。而有大的参考书，或者“动物”书，提供全面的材料，覆盖了整个的课题。两种书都有优缺点，因此只出版一种书是有问题的。

为了阐明这个问题，考虑如下情形，数万的读者读完开发者笔记后，他们都知道了如何创建一个简单的项目，比方说一个Maven项目从一组源文件创建一个WAR。但是，当他们想知道更多的细节或者类似于Assembly插件的详细描述的时候，他们会陷入僵局。因为没有关于Maven的写得很好的参考手册，他们需要在Maven站点上搜寻插件文档，或者在一系列邮件列表中不停挑选。当人们真正开始钻研Maven的时候，他们开始在Maven站点上阅读无数的由很多不同的开发人员编写的粗糙的HTML文档，这些开发人员对为插件编写文档有着完全不同的想法：数百的的开发人员有着不同的书写风格，语气以其方言。除去很多好心的志愿者所做的努力，在Maven站点上阅读插件文档，最好的情况下，令人有受挫感，最坏的情况下，成为了抛弃Maven的理由。很多情况下Maven用户感觉“被骗了”因为他们不能从站点文档上找到答案。

虽然第一本Maven开发者笔记吸引了很多Maven的新用户，为很多人培训了最基本的Maven用例，但同样多的读者，当他们不能找到准确的写得很好的参考材料的时候，感觉到了挫败感。很多年来，缺少可靠的（或者权威的）参考资料阻碍了Maven的发展；这也成了一种Maven用户社区成长的阻碍力量。本书想要改变这种情况，通过提供，第一：在第 I 部分 “Maven实战”中更新原本的Maven开发者笔记，第二：在第 II 部分 “Maven参考”中第一次尝试提供全面的参考。在你手里的（或者屏幕上的）实际上是二书合一。

本书的这个部分中，我们不会抛弃开发者笔记中的描述性推进方式，这是帮助人们“以实战方式”学习Maven的很有价值的材料。在本书的第一部分中我们“边做边介绍”，然后在第 II 部分 “Maven参考”中我们填充空白，钻研细节，介绍那些Maven新手不感兴趣的高级话题。第 II 部分 “Maven参考”可能使用与样例项目无关的一个参考列表和一程序列表，而第 I 部分 “Maven实战”将由实际样例和故事驱动。读完第 I 部分 “Maven实战”后，你将会拥有几个月内开始使用Maven所需要的一切。只有当你需要通过编写定制插件来开始自定义你的项目，或者想了解特定插件细节的时候，才可能需要回到第 II 部分 “Maven参考”。

[1] http://www.oreilly.com

[2] http://www.oreilly.com/catalog/9780596007508/

第 3 章 一个简单的Maven项目

3.1. 简介

本章我们介绍一个用Maven Archetype插件从空白开始创建的简单项目。 当你跟着这个简单项目的开发过程，你会看到这个简单的应用给我们提供了介绍Maven核心概念的机会。

在你能开始使用Maven做复杂的，多模块的构建之前，我们需要从基础开始。如果你之前用过Maven，你将会注意到这里很好的照顾到了细节。你的构建倾向于“只要能工作”，只有当你需要编写插件来自定义默认行为的时候，才需要深入Maven的细节。另一方面，当你需要深入Maven细节的时候，对Maven核心概念的彻底理解是至关重要的。本章致力于为你介绍一个尽可能简单的Maven项目，然后介绍一些使Maven成为一个可靠的构建平台的核心概念。读完本章后，你将会对构建生命周期 (build lifecycle)，Maven仓库 (repositories)，依赖管理 (dependency management)和项目对象模型 (Project Object Model)有一个基本的理解。

3.1.1. 下载本章的例子

本章开发了一个十分简单的例子，它将被用来探究Maven的核心概念。如果你跟着本章表述的步骤，你应该不需要下载这些例子来重新创建那些Maven已经生成好的代码。我们将会使用Maven Archetype插件来创建这个简单的项目，本章不会以任何方式修改这个项目。如果你更喜欢通过最终的例子源代码来阅读本章，本章的例子项目和这本书的例子代码可以从这里下载到：http://www.sonatype.com/book/mvn-examples-1.0.zip或者http://www.sonatype.com/book/mvn-examples-1.0.tar.gz。解压存档文件到任何目录下，然后到ch03/目录。在ch03/目录你将看到一个名字为simple/的目录，它包含了本章的源代码。如果你希望在Web浏览器里看这些例子代码，访问http://www.sonatype.com/book/examples-1.0并且点击ch03/目录。

3.2. 创建一个简单的项目

开始一个新的Maven项目，在命令行使用Maven Archetype插件。

```
$ mvn archetype:create -DgroupId=org.sonatype.mavenbook.ch03 \
                                           -DartifactId=simple \
                                           -DpackageName=org.sonatype.mavenbook
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'archetype'.
[INFO] artifact org.apache.maven.plugins:maven-archetype-plugin: checking for \
       updates from central
[INFO] ------------------------------------------------------------------------
[INFO] Building Maven Default Project
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛


```
[INFO]    task-segment: [archetype:create] (aggregator-style)
[INFO] ------------------------------------------------------------------------
[INFO] [archetype:create]
[INFO] artifact org.apache.maven.archetypes:maven-archetype-quickstart: \
       checking for updates from central
[INFO] Parameter: groupId, Value: org.sonatype.mavenbook.ch03
[INFO] Parameter: packageName, Value: org.sonatype.mavenbook
[INFO] Parameter: basedir, Value: /Users/tobrien/svnw/sonatype/examples
[INFO] Parameter: package, Value: org.sonatype.mavenbook
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] Parameter: artifactId, Value: simple
[INFO] * End of debug info from resources from generated POM *
[INFO] Archetype created in dir: /Users/tobrien/svnw/sonatype/examples/simple
```

mvn 是Maven2的命令。archetype:create称为一个Maven目标 (goal)。如果你熟悉Apache Ant，一个Maven目标类似于一个Ant目标 (target)；它们都描述了将会在构建中完成的工作单元 (unit of work)。而像-Dname=value这样的对是将会被传到目标中的参数，它们使用-D属性这样的形式[1]，类似于你通过命令行向Java虚拟机传递系统属性。archetype:create这个目标的目的通过archetype快速创建一个项目。在这里，一个archetype被定义为“一个原始的模型或者类型，在它之后其它类似的东西与之匹配；一个原型(prototype)”。Maven有许多可用的archetype，从生成一个简单的Swing应用，到一个复杂的Web应用。本章我们用最基本的archetype来创建一个入门项目的骨架。这个插件的前缀是“archetype”，目标为”create”。[1]

我们已经生成了一个项目，看一下Maven在simple目录下创建的目录结构：

```
simple/❶
simple/pom.xml❷
      /src/
      /src/main/❸
          /main/java
      /src/test/❹
          /test/java
```

这个生成的目录遵循Maven标准目录布局，我们之后会去看更多的细节，但是，现在让我们只是尝试了解这些基本的目录。

❶ Maven Archtype插件创建了一个与artifactId匹配的目录——simple。这是项目的基础目录。

❷ 每个项目在文件pom.xml里有它的项目对象模型 (POM)。这个文件描述了这个项目，配置了插件，声明了依赖。

[1]”-D<name>=<value>”这种格式不是Maven定义的，它其实是Java用来设置系统属性的方式，可以通过“java -help”查看Java的解释。Maven的bin目录下的脚本文件仅仅是把属性传入Java而已。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

❸ 我们项目的源码了资源文件被放在了src/main目录下面。在我们简单Java项目这样的情况下，这个目录包含了一下java类和一些配置文件。在其它的项目中，它可能是web应用的文档根目录，或者还放一些应用服务器的配置文件。在一个Java项目中，Java类放在src/main/java下面，而classpath资源文件放在src/main/resources下面。

❹ 我们项目的测试用例放在src/test下。在这个目录下面，src/test/java存放像使用JUnit或者TestNG这样的Java测试类。目录src/test/resources下存放测试classpath资源文件。

Maven Archtype插件生成了一个简单的类org.sonatype.mavenbook.App，它是一个仅有13行代码的Java，所做的只是在main方法中输出一行消息：

```
package org.sonatype.mavenbook;

/**
 * Hello world!
 *
 */
public class App
{
    public static void main( String[] args )
    {
        System.out.println( "Hello World!" );
    }
}
```

最简单的Maven archetype生成最简单的Maven项目：一个往标准输出打印“Hello World”的程序。

3.3. 构建一个简单的项目

一旦你遵循第 3.2 节 “创建一个简单的项目使用Maven Archetype插件创建了一个项目，你会希望构建并打包这个应用。想要构建打包这个应用，在包含pom.xml的目录下运行mvn install。

```
$ mvn install
[INFO] Scanning for projects...
[INFO] ----------------------------------------------------------------------------
[INFO] Building simple
[INFO]    task-segment: [install]
[INFO] ----------------------------------------------------------------------------
[INFO] [resources:resources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:compile]
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛


```
[INFO] Compiling 1 source file to /simple/target/classes
[INFO] [resources:testResources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:testCompile]
[INFO] Compiling 1 source file to /simple/target/test-classes
[INFO] [surefire:test]
[INFO] Surefire report directory: /simple/target/surefire-reports

-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running org.sonatype.mavenbook.AppTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.105 sec

Results :

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0

[INFO] [jar:jar]
[INFO] Building jar: /simple/target/simple-1.0-SNAPSHOT.jar
[INFO] [install:install]
[INFO] Installing /simple/target/simple-1.0-SNAPSHOT.jar to \
  ~/.m2/repository/org/sonatype/mavenbook/ch03/simple/1.0-SNAPSHOT/ \
  simple-1.0-SNAPSHOT.jar
```

你已经创建了，编译了，测试了，打包了，并且安装了(installed)最简单的Maven项目。在命令行运行它以向你自己验证这个程序能工作。

```
$ java -cp target/simple-1.0-SNAPSHOT.jar org.sonatype.mavenbook.App
Hello World!
```

3.4. 简单的项目对象模型 (Project Object Model)

当Maven运行的时候它向项目对象模型(POM)查看关于这个项目的信息。POM回答类似这样的问题：这个项目是什么类型的？这个项目的名称是什么？这个项目的构建有自定义么？这里是一个由Maven Archetype插件的create目标创建的默认的pom.xml文件。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

例 3.1. Simple 项目的pom.xml 文件

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/mave
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.sonatype.mavenbook.ch03</groupId>
  <artifactId>simple</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>simple</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

这个pom.xml文件是你将会面对的Maven项目中最基础的POM，一般来说一个POM文件会复杂得多：定义多个依赖，自定义插件行为。最开始的几个元素——groupId，artifactId，packaging，version——是Maven的坐标(coordinates)，它们唯一标识了一个项目。name和url是POM提供的描述性元素，它们给人提供了可阅读的名字，将一个项目关联到了项目web站点。最后，dependencies元素定义了一个单独的，测试范围(test-scoped)依赖，依赖于称为JUnit的单元测试框架。这些话题将会第 3.5 节 “核心概念被深入介绍，当前，你所需知道的是，pom.xml是一个让Maven跑起来的文件。

当Maven运行的时候，它是根据项目的pom.xml里设置的组合来运行的，一个最上级的POM定义了Maven的安装目录，在这个目录中全局的默认值被定义了，（可能）还有一些用户定义的设置。想要看这个“有效的 (effective)” POM，或者说Maven真正运行根据的POM，在simple项目的基础目录下跑下面的命令。

```
$ mvn help:effective-pom
```

一旦你运行了此命令，你应该能看到一个大得多的POM，它暴露了Maven的默认设置

3.5. 核心概念

我们已经第一次运行了Maven，是时候介绍一些Maven的核心概念了。在之前的例子中，我们生了一个项目，它包含了一个POM和一些源代码，它们一起组成了Maven的标准目录

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

布局。之后你用生命周期阶段(phase)作为参数来运行Maven，这个阶段会提示Maven运行一系列Maven插件的目标。最后，你把Maven构件(artifact)安装(install)到了你本地仓库(repository)。等等？什么是生命周期？什么是“本地仓库”？下面的小结阐述了一些Maven的核心概念。

3.5.1. Maven插件和目标 (Plugins and Goals)

在前面一节中，我们用两种类型的命令行参数运行了Maven。第一条命令是一条单个的插件目标，Archetype插件的create目标。Maven第二次运行是一个生命周期阶段 – install。为了运行单个的Maven插件目标，我们使用mvn archetype:create这样的语法，这里archetype是一个插件标识而create是目标标识。当Maven运行一个插件目标，它向标准输出打印出插件标识和目标标识：

```
$ mvn archetype:create -DgroupId=org.sonatype.mavenbook.ch03 \
                                        -DartifactId=simple \
                                        -DpackageName=org.sonatype.mavenbook
...
[INFO] [archetype:create]
[INFO] artifact org.apache.maven.archetypes:maven-archetype-quickstart: \
       checking for updates from central
...
```

一个Maven插件是一个单个或者多个目标的集合。Maven插件的例子有一些简单但核心的插件，像Jar插件，它包含了一组创建JAR文件的目标，Compiler插件，它包含了一组编译源代码和测试代码的目标，或者Surefire插件，它包含一组运行单元测试和生成测试报告的目标。而其它的，更有专门的插件包括：Hibernate3插件，用来集成流行的持久化框架Hibernate，JRuby插件，它让你能够让运行ruby称为Maven构建的一部分或者用Ruby来编写Maven插件。Maven也提供了自定义插件的能力。一个定制的插件可以用Java编写，或者用一些其它的语言如Ant，Groovy，beanshell和之前提到的Ruby。

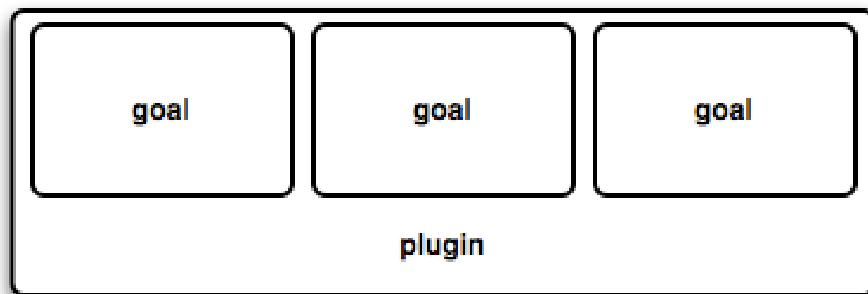


图 3.1. 一个插件包含一些目标

一个目标是一个明确的任务，它可以作为单独的目标运行，或者作为一个大的构建的一部分和其它目标一起运行。一个目标是Maven中的一个“工作单元(unit of

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

work)”。目标的例子包括Compiler插件中的compile目标，它用来编译项目中的所有源文件，或者Surefire插件中的test目标，用来运行单元测试。目标通过配置属性进行配置，以用来定制行为。例如，Compiler插件的compile目标定义了一组配置参数，它们允许你设置目标JDK版本或者选择是否用编译优化。在之前的例子中，我们通过命令行参数-DgroupId=org.sonatype.mavenbook.ch03和-DartifactId=simple向Archetype插件的create目标传入了groupId和artifactId配置参数。我们也向create目标传入了packageName参数，它的值为org.sonatype.mavenbook。如果我们忽略了packageName参数，那么包名的默认值为org.sonatype.mavenbook.ch03。

注意

当提到一个插件目标的时候，我们常常用速记符号：pluginId:goalId。例如，当提到Archetype插件的create目标的时候，我们写成archetype:create。

目标定义了一些参数，这些参数可以定义一些明智的默认值。在archetype:create这个例子中，我们并没有在命令行中指定这个目标创建什么类型的archetype，我们简单的传入一个groupId和一个artifactId。这是我们对于约定优于配置(convention over configuration)的第一笔。这里create目标的约定，或者默认值，是创建一个简单的项目，叫做Quickstart。create目标定义了一个配置属性archetypeArtifactId，它有一个默认值为maven-archetype-quickstart。Quickstart archetype生成了一个最小项目的躯壳，包括一个POM和一个类。Archetype插件比第一个例子中的样子强大得多，但是这是一个快速开始新项目的不错的方法。在本书的后面，我们将会让你看到Archetype插件可以用来生成复杂如web应用的项目，以及你如何能够使用Archetype插件来定义你自己项目的集合。

Maven的核心对你项目构建中特定的任务几乎毫无所知。就它本身来说，Maven不知道如何编译你的代码，它甚至不知道如何制作一个JAR文件，它把所有这些任务代理给了Maven插件，像Compiler插件和Jar插件，它们在需要的时候被下载下来并且定时的从Maven中央仓库更新。当你下载Maven的时候，你得到的是一个包含了基本躯壳的Maven核心，它知道如何解析命令行，管理classpath，解析POM文件，在需要的时候下载Maven插件。通过保持Compiler插件和Maven核心分离，并且提供更新机制，用户很容易能使用编译器最新的版本。通过这种方式，Maven插件提供了通用构建逻辑的全局重用性，有不会在构建周期中定义编译任务，有使用了所有Maven用户共享的Compiler插件。如果有个对Compiler插件的改进，每个使用Maven的项目可以立刻从这种变化中得到好处。（并且，如果你不喜欢这个Compiler插件，你可以用你的实现来覆盖它）。

3.5.2. Maven生命周期 (Lifecycle)

上一节中，我们运行的第二个命令是mvn package。命令行并没有指定一个插件目标，而是指定了一个Maven生命周期阶段。一个阶段是在被Maven称为“构建生命周期”中的一个步骤。生命周期是包含在一个项目构建中的一系列有序的阶段。Maven可以支

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

持许多不同的生命周期，但是最常用的生命周期是默认的Maven生命周期，这个生命周期中一开始的一个阶段是验证项目的基本完整性，最后的一个阶段是把一个项目发布成产品。生命周期的阶段被特地留得含糊，单独的定义为验证(validation)，测试(testing)，或者发布(deployment)，而他们对不同项目来说意味着不同的事情。例如，打包(package)这个阶段在一个项目里生成一个JAR，它也就意味着“将一个项目打成一个jar”，而在另外一个项目里，打包这个阶段可能生成一个WAR文件。图 3.2 “一个生命周期是一些阶段的序列”展示了默认Maven生命周期的简单样子。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

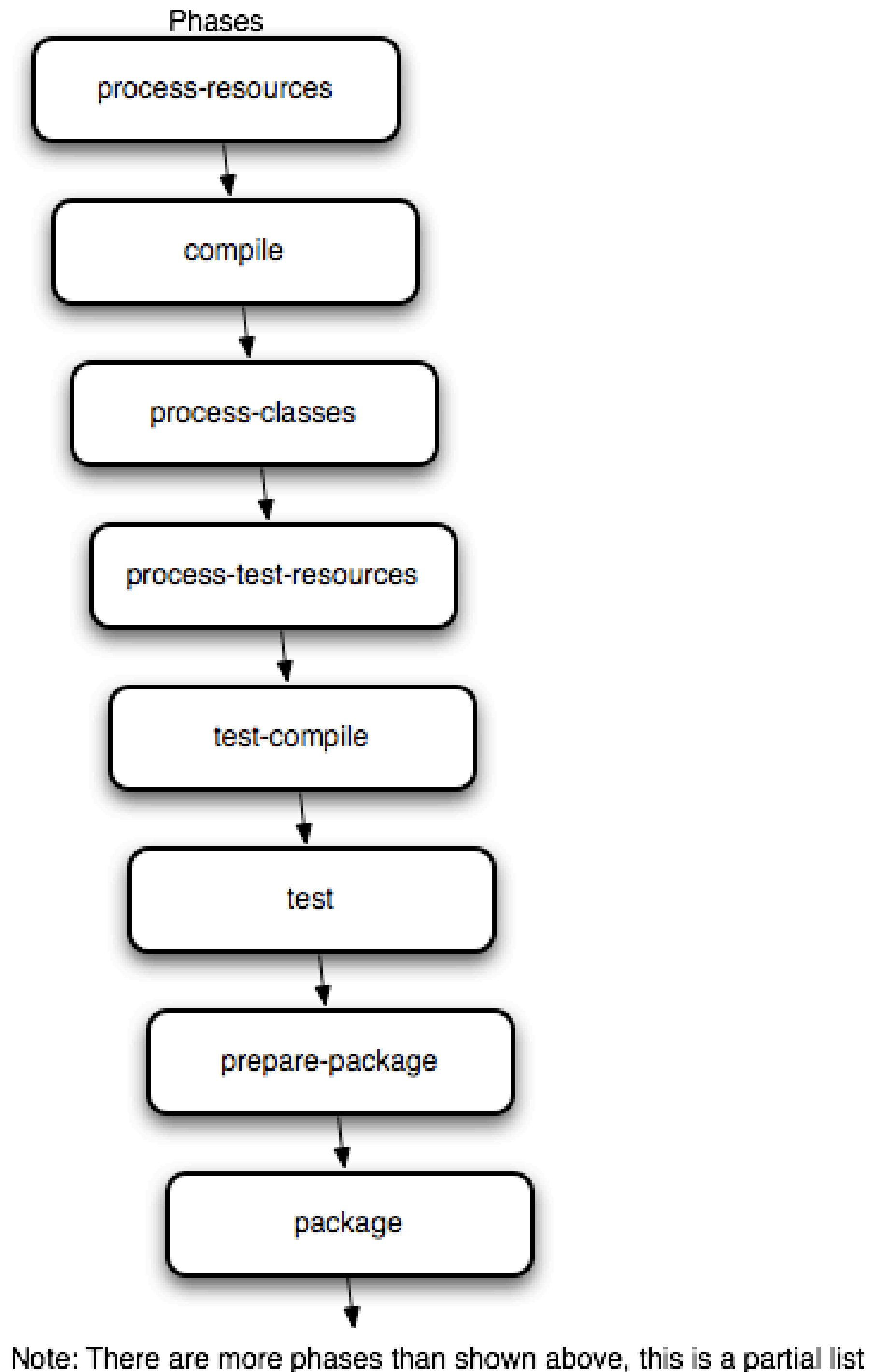


图　3.2.　一个生命周期是一些阶段的序列

插件目标可以附着在生命周期阶段上。随着Maven沿着生命周期的阶段移动，它会执行附着在特定阶段上的目标。每个阶段可能绑定了零个或者多个目标。在之前的小节

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

里，当你运行mvn package，你可能已经注意到了不止一个目标被执行了。检查运行mvn package之后的输出，会注意到那些被运行的各种目标。当这个简单例子到达package阶段的时候，它运行了Jar插件的jar目标。既然我们的简单的quickstart项目（默认）是jar包类型，jar:jar目标被就绑定到了打包阶段。

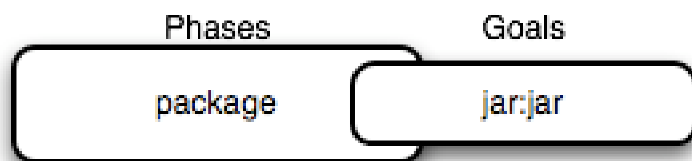


图 3.3. 一个目标绑定到一个阶段

我们知道，在包类型为jar的项目中，打包阶段将会创建一个JAR文件。但是，在它之前的目标做什么呢，像compiler:compile和surefire:test？在Maven经过它生命周期中package之前的阶段的时候，这些目标被运行了；Maven执行一个阶段的时候，它首先会有序的执行前面的所有阶段，到命令行指定的那个阶段为止。每个阶段对应了零个或者多个目标。我们没有进行任何插件配置或者定制，所以这个例子绑定了一组标准插件的目标到默认的生命周期。当Maven经过以package为结尾的默认生命周期的时候，下面的目标按顺序被执行：

`resources:resources`
: Resources插件的resources目标绑定到了resources 阶段。这个目标复制src/main/resources下的所有资源和其它任何配置的资源目录，到输出目录。

`compiler:compile`
: Compiler插件的compile目标绑定到了compile 阶段。这个目标编译src/main/java下的所有源代码和其他任何配置的资源目录，到输出目录。

`resources:testResources`
: Resources插件的testResources目标绑定到了test-resources 阶段。这个目标复制src/test/resources下的所有资源和其它任何的配置的测试资源目录，到测试输出目录。

`compiler:testCompile`
: Compiler插件的testCompile目标绑定到了test-compile 阶段。这个目标编译src/test/java下的测试用例和其它任何的配置的测试资源目录，到测试输出目录。

`surefire:test`
: Surefire插件的test目标绑定到了test 阶段。这个目标运行所有的测试并且创建那些捕捉详细测试结果的输出文件。默认情况下，如果有测试失败，这个目标会终止。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

`jar:jar`
: Jar插件的`jar`目标绑定到了`package` 阶段。这个目标把输出目录打包成JAR文件。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

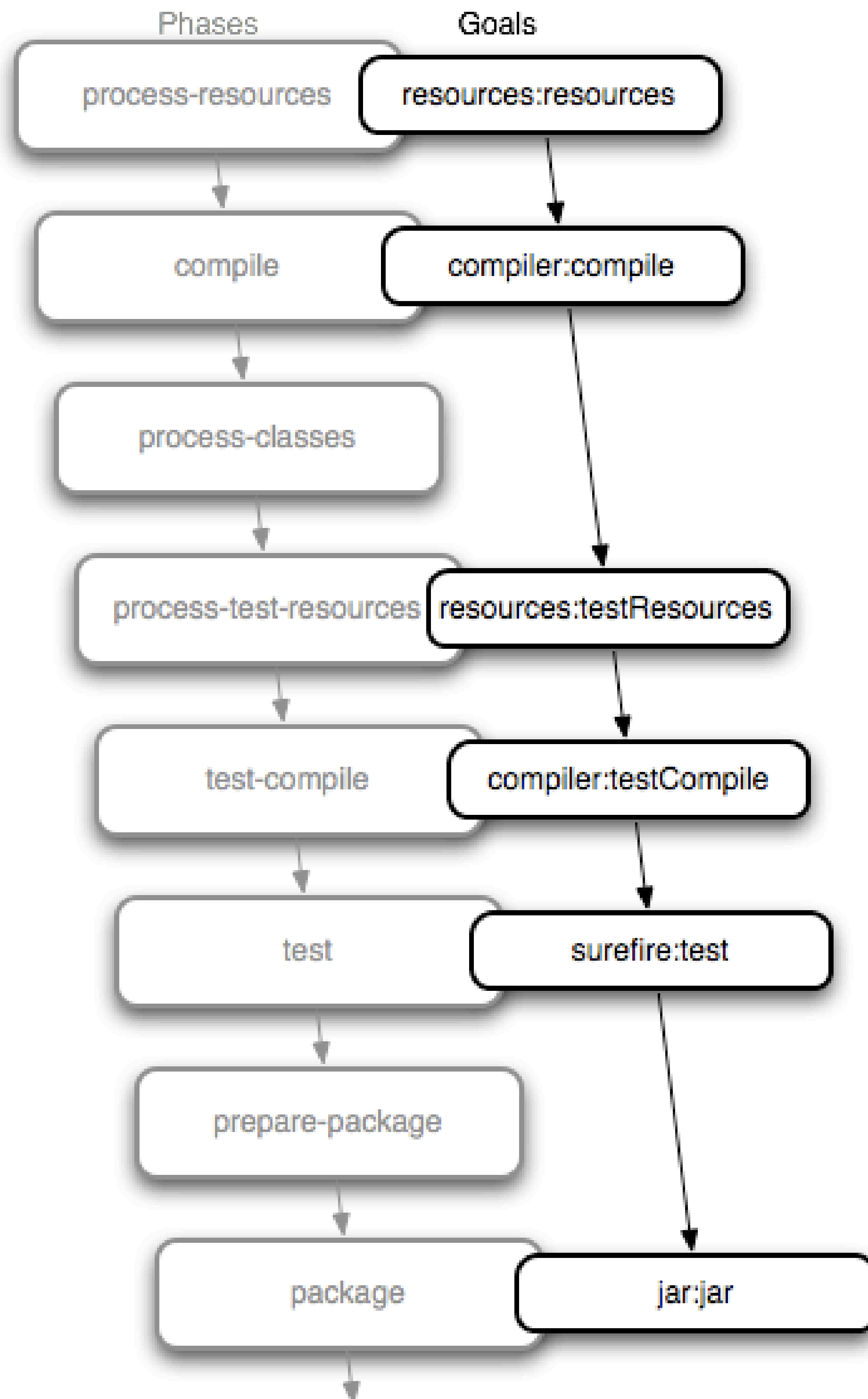


图　3.4.　被绑定的目标随着它们阶段的运行而运行

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

总结得来说，当我们运行mvn package，Maven运行到打包为止的所有阶段，在Maven沿着生命周期一步步向前的过程中，它运行绑定在每个阶段上的所有目标。你也可以像下面这样显式的指定一系列插件目标，以得到同样的结果：

```
mvn resources:resources \
    compiler:compile \
    resources:testResources \
    compiler:testCompile \
    surefire:test \
    jar:jar
```

运行package阶段能很好的跟踪一个特定的构建中包含的所有目标，它也允许每个项目使用Maven来遵循一组定义明确的标准。而这个生命周期能让开发人员从一个Maven项目跳到另外一个Maven项目，而不用知道太多每个项目构建的细节。如果你能够构建一个Maven项目，那么你就能构建所有的Maven项目。

3.5.3. Maven坐标 (Coordinates)

Archetype插件通过名字为pom.xml的文件创建了一个项目。这就是项目对象模型（POM），一个项目的声明性描述。当Maven运行一个目标的时候，每个目标都会访问定义在项目POM里的信息。当jar:jar目标需要创建一个JAR文件的时候，它通过观察POM来找出这个Jar文件的名字。当compiler:compile任务编译Java源代码为字节码的时候，它通过观察POM来看是否有编译目标的参数。目标在POM的上下文中运行。目标是我们希望针对项目运行的动作，而项目是通过POM定义的。POM为项目命名，提供了项目的一组唯一标识符（坐标），并且通过依赖 (dependencies) ，父 (parents) 和先决条件(prerequisite) 来定义和其它项目的关系。POM也可以自定义插件行为，提供项目相关的社区和开发人员的信息。

Maven坐标定义了一组标识，它们可以用来唯一标识一个项目，一个依赖，或者Maven POM里的一个插件。看一下下面的POM。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛


```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>mavenbook</groupId>
  <artifactId>my-app</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>Maven Quick Start Archetype</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

coordinates

图 3.5. 一个Maven项目的坐标

我们加亮了这个项目的坐标：groupId，artifactId，version和packaging。这些组合的标识符拼成了一个项目的坐标[2]。[2]就像任何其它的坐标系统，一个Maven坐标是一个地址，即“空间”里的某个点：从一般到特殊。当一个项目通过依赖，插件或者父项目引用和另外一个项目关联的时候，Maven通过坐标来精确定位一个项目。Maven坐标通常用冒号来作为分隔符来书写，像这样的格式：groupId:artifactId:packaging:version。在上面的pom.xml中，它的坐标可以表示为mavenbook:my-app:jar:1.0-SNAPSHOT. 这个符号也适用于项目依赖，我们的项目依赖JUnit的3.8.1版本，它包含了一个对junit:junit:jar:3.8.1的依赖。

groupId
: d 团体，公司，小组，组织，项目，或者其它团体。团体标识的约定是，它以创建这个项目的组织名称的逆向域名(reverse domain name)开头。来自Sonatype的项目有一个以com.sonatype开头的groupId，而Apache Software的项目有以org.apache开头的groupId。

artifactId
: 在groupId下的表示一个单独项目的唯一标识符。

[2]还有第五个，名为classifier的很少使用的坐标，将在本书后面介绍。现在你尽管可以忽略classifiers。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

`version`
: 一个项目的特定版本。发布的项目有一个固定的版本标识来指向该项目的某一个特定的版本。而正在开发中的项目可以用一个特殊的标识，这种标识给版本加上一个“SNAPSHOT”的标记。

项目的打包格式也是Maven坐标的重要组成部分，但是它不是项目唯一标识符的一个部分。一个项目的`groupId:artifactId:version`使之成为一个独一无二的项目；你不能同时有一个拥有同样的`groupId`，`artifactId`和`version`标识的项目。

`packaging`
: 项目的类型，默认是`jar`，描述了项目打包后的输出。类型为`jar`的项目产生一个JAR文件，类型为`war`的项目产生一个web应用。

在其它“Maven化”项目构成的巨大空间中，的这四个元素是定位和使用某个特定项目的关键因素。Maven仓库(repositories)（公共的，私有的，和本地的）是通过这些标识符来组织的。当一个项目被安装到本地的Maven仓库，它立刻能被任何其它的项目所使用。而我们所需要做的只是，在其它项目用使用Maven的唯一坐标来加入对这个特定构件的依赖。

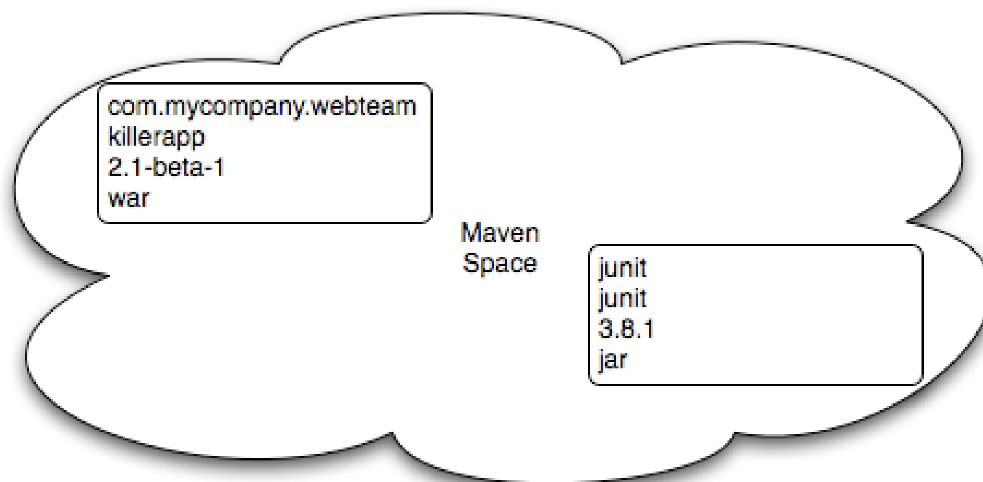


图 3.6. Maven空间是项目的一个坐标系统

3.5.4. Maven仓库(Repositories)

当你第一次运行Maven的时候，你会注意到Maven从一个远程的Maven仓库下载了许多文件。如果这个简单的项目是你第一次运行Maven，那么当触发resources:resource目标的时候，它首先会做的事情是去下载最新版本的Resources插件。在Maven中，构件和

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

插件是在它们被需要的时候从远程的仓库取来的。初始的Maven下载包的大小相当的小（1.8兆），其中一个原因是事实上这个初始Maven不包括很多插件。它只包含了几近赤裸的最少值，而在需要的时候再从远程仓库去取。Maven自带了一个用来下载Maven核心插件和依赖的远程仓库地址（http://repo1.maven.org/maven2）。

你常常会写这样一个项目，这个项目依赖于一些既不免费也不公开的包。在这种情况下，你需要要么在你组织的网络里安装一个定制的仓库，要么手动的安装这些依赖。默认的远程仓库可以被替换，或者增加一个你组织维护的自定义Maven仓库的引用。有许多现成的项目允许组织管理和维护公共Maven仓库的镜像。

是什么让Maven仓库成为一个Maven仓库的呢？Maven仓库是通过结构来定义的，一个Maven仓库是项目构件的一个集合，这些构件存储在一个目录结构下面，它们的格式能很容易的被Maven所理解。在一个Maven仓库中，所有的东西存储在一个与Maven项目坐标十分匹配的目录结构中。你可以打开浏览器，然后浏览中央Maven仓库http://repo1.maven.org/maven2/ 来看这样的结构。你会看到坐标为`org.apache.commons:commons-email:1.1`的构件能在目录`/org/apache/commons/commons-email/1.1/`下找到，文件名为`commons-email-1.1.jar`。Maven仓库的标准是按照下面的目录格式来存储构件，相对于仓库的根目录：

```
/<groupId>/<artifactId>/<version>/<artifactId>-<version>.<packaging>
```

Maven从远程仓库下载构件和插件到你本机上，存储在你的本地Maven仓库里。一旦Maven已经从远程仓库下载了一个构件，它将永远不需要再下载一次，因为maven会首先在本地仓库查找插件，然后才是其它地方。在Windows XP上，你的本地仓库很可能在`C:\Documents and Settings\USERNAME\.m2\repository`，在Windows Vista上，会是`C:\Users\USERNAME\.m2\repository`。在Unix系统上，你的本地仓库在`~/.m2/repository`。当你创建像前一节创建的简单项目时，`install`阶段执行一个目标，把你项目的构件安装到你的本地仓库。

在你的本地仓库，你应该可以看到我们的简单项目创建出来的构件。如果你运行mvn install命令，Maven会把我们项目的构件安装到本地仓库。试一下。

```
$ mvn install
...
[INFO] [install:install]
[INFO] Installing .../simple-1.0-SNAPSHOT.jar to \
       ~/.m2/repository/org/sonatype/mavenbook/simple/1.0-SNAPSHOT/ \
       simple-1.0-SNAPSHOT.jar
...
```

就像你能从这个命令的输出看到的，Maven把我们项目的JAR文件安装到了我们的本地Maven仓库。Maven在本地项目中通过本地仓库来共享依赖。如果你开发了两个项目——项目A和项目B——项目B依赖于项目A产生的构件。当构建项目B的时候，Maven会从本地

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

仓库取得项目A的构件。Maven仓库既是一个从远程仓库下载的构件的缓存，也允许你的项目相互依赖。

3.5.5. Maven依赖管理 (Dependency Management)

在本章的simple样例中，Maven处理了JUnit依赖的坐标——`junit:junit:3.8.1`，指向本地Maven仓库中的`/junit/junit/3.8.1/junit-3.8.1.jar`。这种基于Maven坐标的定位构件的能力能让我们在项目的POM中定义依赖。如果你检查simple项目的`pom.xml`文件，你会看到有一个文件中有一个段专门处理`dependencies`，那里面包含了一个单独的依赖——JUnit。

一个复杂的项目将会包含很多依赖，也有可能包含依赖于其它构件的依赖。这是Maven最强大的特征之一，它支持了传递性依赖（transitive dependencies）。假如你的项目依赖于一个库，而这个库又依赖于五个或者十个其它的库（就像Spring或者Hibernate那样）。你不必找出所有这些依赖然后把它们写在你的`pom.xml`里，你只需要加上你直接依赖的那些库，Maven会隐式的把这些库间接依赖的库也加入到你的项目中。Maven也会处理这些依赖中的冲突，同时能让你自定义默认行为，或者排除一些特定的传递性依赖。

让我们看一下你运行前面的样例的时候那些下载到你本地仓库的依赖。看一下这个目录：`~/.m2/repository/junit/junit/3.8.1/`。如果你一直跟着本章的样例，那么这里会有文件`junit-3.8.1.jar` 和`junit-3.8.1.pom`，还有Maven用来验证已下载构件准确性的校验和文件。需要注意的是Maven不只是下载JUnit的JAR文件，它同时为这个JUnit依赖下载了一个POM文件。Maven同时下载构件和POM文件的这种行为，对Maven支持传递性依赖来说非常重要。

当你把项目的构件安装到本地仓库时，你会发现在和JAR文件同一目录下，Maven发布了一个稍微修改过的`pom.xml`的版本。存储POM文件在仓库里提供给其它项目了该项目的信息，其中最重要的就是它有哪些依赖。如果项目B依赖于项目A，那么它也依赖于项目A的依赖。当Maven通过一组Maven坐标来处理依赖构件的时候，它也会获取POM，通依赖的POM来寻找传递性依赖。那些传递性依赖就会被添加到当前项目的依赖列表中。

在Maven中一个依赖不仅仅是一个JAR。它是一个POM文件，这个POM可能也声明了对其它构件的依赖。这些依赖的依赖叫做传递性依赖，Maven仓库不仅仅存贮二进制文件，也存储了这些构建的元数据（metadata），才使传递性依赖成为可能。下图展现了一个传递性依赖的可能场景。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

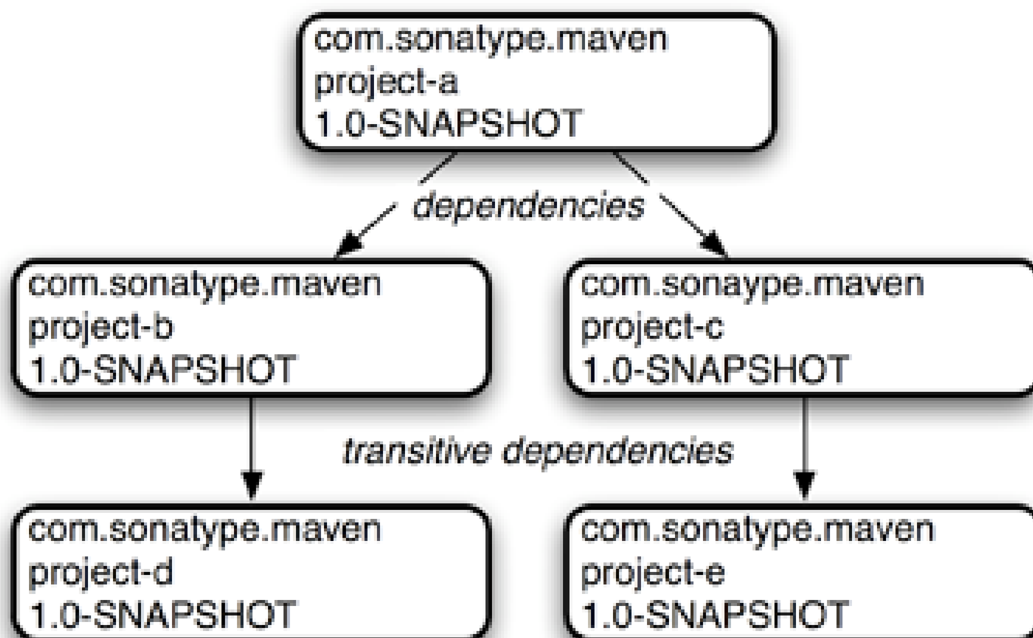


图　3.7.　Maven处理传递性依赖

在上图中，项目A依赖于项目B和C，项目B依赖于项目D，项目C依赖于项目E，但是项目A所需要做的只是定义对B和C的依赖。当你的项目依赖于其它的项目，而这些项目又有一些小的依赖时(向Hibernate, Apache Struts 或者 Spring Framework)，传递性依赖使之变得相当的方便。Maven同时也提供了一种机制，能让你排除一些你不想要的传递性依赖。

Maven也提供了不同的依赖范围(dependency scope)。Simple项目的`pom.xml`包含了一个依赖——`junit:junit:jar:3.8.1`——范围是`test`。当一个依赖的范围是`test`的时候，说明它在Compiler插件运行`compile`目标的时候是不可用的。它只有在运行`compiler:testCompile`和`surefire:test`目标的时候才会被加入到classpath中。

当为项目创建JAR文件的时候，它的依赖不会被捆绑在生成的构件中，他们只是用来编译。当用Maven来创建WAR或者EAR，你可以配置Maven让它在生成的构件中捆绑依赖，你也可以配置Maven，使用`provided`范围，让它排除WAR文件中特定的依赖。`provided`范围告诉Maven一个依赖在编译的时候需要，但是它不应该被捆绑在构建的输出中。当你开发web应用的时候`provided`范围变得十分有用，你需要通过Servlet API来编译你的代码，但是你不希望Servlet API的JAR文件包含在你web应用的`WEB-INF/lib`目录中。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

3.5.6. 站点生成和报告 (Site Generation and Reporting)

另外一个Maven的重要特征是，它能生成文档和报告。在simple项目的目录下，运行以下命令：

```
$ mvn site
```

这将会运行site生命周期阶段。它不像默认生命周期那样，管理代码生成，操作资源，编译，打包等等。Site生命周期只关心处理在src/site目录下的site内容，还有生成报告。在这个命令运行过之后，你将会在target/site目录下看到一个项目web站点。载入target/site/index.html你会看到项目站点的基本外貌。它包含了一些报告，它们在左手边的导航目录的“项目报告”下面。它也包含了项目相关的信息，依赖和相关开发人员信息，在“项目信息”下面。Simple项目的web站点大部分是空的，因为POM只包含了比较少的信息，只有项目坐标，名称，URL和一个test依赖。

在这个站点上，你会注意到一些默认的报告已经可以访问了，有一个报告详细描述了测试的结果。这个单元测试报告描述了项目中所有单元测试的成功和失败信息。另外一个报告生成了项目API的JavaDoc。Maven提供了很完整的可配置的报告，像Clover报告检查单元测试覆盖率，JXR报告生成HTML源代码相互间引用，这在代码审查的时候非常有用，PMD报告针对各种编码问题来分析源代码，JDepend报告分析源代码中各个包之间的依赖。通过在pom.xml中配置那些报告被包含在构建中，站点报告就可以被定制了。

3.6. 小结

我们创建了一个simple项目，将其打包为一个Jar，安装到了Maven仓库使之能被其它项目使用，最后生成了带有文档的站点。不写一行代码，不碰一个配置文件，我们就做到了这些。我们花了一些时间来研究Maven核心概念的定义。在下一章，我们将会自定义并修改我们项目的pom.xml文件，加入一些依赖，配置一些单元测试。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

第 4 章 定制一个Maven项目

4.1. 介绍

本章在上一章所介绍信息的基础上进行开发。 你将创建一个由 Maven Archetype 插件生成的项目，添加一些依赖和一些源代码，并且根据你的需要定制项目。本章最后，你将知道如何使用 Maven 开始创建真正的项目。

4.1.1. 下载本章样例

本章我们将开发一个和 Yahoo! Weather web 服务交互的实用程序。虽然没有样例源码你也应该能够理解这个开发过程，但还是推荐你下载本章样例源码以作为参考。 本章的样例项目包含在本书的样例代码中，你可以从两个地方下载，http://www.sonatype.com/book/mvn-examples-1.0.zip 或者 http://www.sonatype.com/book/mvn-examples-1.0.tar.gz 。解压存档文件至任意目录，然后到 `ch04/` 目录。在 `ch04/` 目录你会看到一个名为 `simple-weather` 的目录，它包含了本章开发出来的 Maven 项目。如果你想要在浏览器里看样例代码，访问 http://www.sonatype.com/book/examples-1.0 ，然后点击 `ch04/` 目录。

4.2. 定义Simple Weather项目

在定制本项目之前，让我们退后一步，讨论下这个 simple weather 项目。 这个 simple weather 项目是什么？ 它是一个被设计成用来示范一些 Maven 特征的样例。它能代表一类你可能需要构建的应用程序。 这个 simple weather 是一个基本的命令行驱动的应用程序，它接受邮政编码输入，然后从 Yahoo! Weather RSS 源获取数据，然后解析数据并把结果打印到标准输出。 我们选择该项目是有许多因素的。 首先，它很直观；用户通过命令行提供输入，程序读取邮政编码，对 Yahoo! Weather 提交请求，之后解析结果，格式化之后输入到屏幕。 这个样例是个简单的 `main()` 函数加上一些相关支持的类；没有企业级框架需要介绍或解释，只有 XML 解析和一些日志语句。 其次，它提供很好的机会来介绍一些有趣的类库，如 Velocity, Dom4j 和 Log4j。 虽然本书集中于 Maven ，但我们不会回避那些介绍有趣工具的机会。 最后，这是一个能在一章内介绍，开发及部署的样例。

4.2.1. Yahoo! Weather RSS

在开始构建这个应用之前，你需要了解一下 Yahoo! Weather RSS 源。该服务是基于以下条款提供的：

> “该数据源免费提供给个人和非营利性组织，作为个人或其它非商业用途。 我们要求你提供给 Yahoo! Weather 连接你数据源应用的权限。”

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

换句话说，如果你考虑集成该数据源到你的商业 web 站点上，请再仔细考虑考虑，该数据源可作为个人或其它非商业性用途。 本章我们提倡的使用是个人教育用途。 要了解更多的 Yahoo! Weather 服务条款，请参考 Yahoo! Weather API 文档： http://developer.yahoo.com/weather/ 。

4.3. 创建Simple Weather项目

首先，让我们用 Maven Archetype 插件创建这个 simple weather 项目的基本轮廓。运行下面的命令，创建新项目：

```
$ mvn archetype:create -DgroupId=org.sonatype.mavenbook.ch04 \
                                          -DartifactId=simple-weather \
                                          -DpackageName=org.sonatype.mavenbook \
                                          -Dversion=1.0
[INFO] [archetype:create]
[INFO] artifact org.apache.maven.archetypes:maven-archetype-quickstart: \
       checking for updates from central
[INFO] ------------------------------------------------------------------
[INFO] Using following parameters for creating Archetype: \
       maven-archetype-quickstart:RELEASE
[INFO] ------------------------------------------------------------------
[INFO] Parameter: groupId, Value: org.sonatype.mavenbook.ch04
[INFO] Parameter: packageName, Value: org.sonatype.mavenbook
[INFO] Parameter: basedir, Value: ~/examples
[INFO] Parameter: package, Value: org.sonatype.mavenbook
[INFO] Parameter: version, Value: 1.0
[INFO] Parameter: artifactId, Value: simple-weather
[INFO] *** End of debug info from resources from generated POM ***
[INFO] Archetype created in dir: ~/examples/simple-weather
```

在 Maven Archetype 插件创建好了这个项目之后，进入到 simple-weather 目录，看一下 `pom.xml`。你会看到如下的 XML 文档：

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

例 4.1. simple-wheather 项目的初始 POM

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                      http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.sonatype.mavenbook.ch04</groupId>
  <artifactId>simple-weather</artifactId>
  <packaging>jar</packaging>
  <version>1.0</version>
  <name>simple-weather2</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

请注意我们给 archetype:create 目标传入了 version 参数。它覆写了默认值 1.0-SNAPSHOT 。本项目中，正如你从 pom.xml 的 version 元素看到的，我们正在开发 simple-weather 项目的 1.0 版本。

4.4. 定制项目信息

在开始编写代码之前，让我们先定制一些项目的信息。 我们想要做的是添加一些关于项目许可证，组织以及项目相关开发人员的一些信息。 这些都是你期望能在大部分项目中看到的标准信息。下面的文档展示了提供组织信息，许可证信息和开发人员信息的 XML 。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

例 4.2. 为 pom.xml 添加组织，法律和开发人员信息

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                      http://maven.apache.org/maven-v4_0_0.xsd">
...

  <name>simple-weather</name>
  <url>http://www.sonatype.com</url>

  <licenses>
    <license>
      <name>Apache 2</name>
      <url>http://www.apache.org/licenses/LICENSE-2.0.txt</url>
      <distribution>repo</distribution>
      <comments>A business-friendly OSS license</comments>
    </license>
  </licenses>

  <organization>
    <name>Sonatype</name>
    <url>http://www.sonatype.com</url>
  </organization>

  <developers>
    <developer>
      <id>jason</id>
      <name>Jason Van Zyl</name>
      <email>jason@maven.org</email>
      <url>http://www.sonatype.com</url>
      <organization>Sonatype</organization>
      <organizationUrl>http://www.sonatype.com</organizationUrl>
      <roles>
        <role>developer</role>
      </roles>
      <timezone>-6</timezone>
    </developer>
  </developers>
...
</project>
```

例 4.2 “为 pom.xml 添加组织，法律和开发人员信息”中的省略号是为了使代码清单变得简短。 当你在 pom.xml 中看到 project 元素的开始标签后面跟着 “…” 或者在 project 元素的结束标签前有 “…” ，这说明我们没有展示整个 pom.xml 文

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

件。在上述情况中，licenses，organization 和 developers 元素是加在 dependencies 元素之前的。

4.5. 添加新的依赖

Simple weather 应用程序必须要完成以下三个任务：从 Yahoo! Weather 获取 XML 数据，解析 XML 数据，打印格式化的输出至标准输出。为了完成这三个任务，我们需要为项目的 pom.xml 引入一些新的依赖。 为了解析来自 Yahoo! 的 XML 响应，我们将会使用 Dom4J 和 Jaxen ，为了格式化这个命令行程序的输出，我们将会使用 Velocity ，我们还需要加入对 Log4j 的依赖，用来做日志。加入这些依赖之后，我们的 dependencies 元素就成了以下模样：

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

例 4.3. 添加 Dom4J, Jaxen, Velocity 和 Log4J 作为依赖

```
<project>
  [...]
  <dependencies>
    <dependency>
      <groupId>log4j</groupId>
      <artifactId>log4j</artifactId>
      <version>1.2.14</version>
    </dependency>
    <dependency>
      <groupId>dom4j</groupId>
      <artifactId>dom4j</artifactId>
      <version>1.6.1</version>
    </dependency>
    <dependency>
      <groupId>jaxen</groupId>
      <artifactId>jaxen</artifactId>
      <version>1.1.1</version>
    </dependency>
    <dependency>
      <groupId>velocity</groupId>
      <artifactId>velocity</artifactId>
      <version>1.5</version>
    </dependency>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
  [...]
</project>
```

*################## test # JUnit ################################ pom.xml ###### **mvn install** ##### Maven ##################### Maven ### ###############“##”### groupId # artifactId ######## (# Log4J) ############################## groupId # artifactId# # Velocity# Dom4J # Jaxen ############ http://www.mvnrepository.com ##### ##### ### Maven #################### ############ http://www.mvnrepository.com ####### ###### Hibernate ## Spring Framework ################### artifactId ### Maven ### ######## ####################################### pom.xml ####### ################ ##### groupId########## mvnrepository.com[1] ####################*

[1] http://www.mvnrepository.com

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

4.6. Simple Weather源码

Simple Weather 命令行应用程序包含五个 Java 类。

`org.sonatype.mavenbook.weather.Main`
: 这个类包含了一个静态的 `main()` 函数，即系统的入口。

`org.sonatype.mavenbook.weather.Weather`
: `Weather` 类是个很简单的 Java Bean，它保存了天气报告的地点和其它一些关键元素，如气温和湿度。

`org.sonatype.mavenbook.weather.YahooRetriever`
: `YahooRetriever` 连接到 Yahoo! Weather 并且返回来自数据源数据的 `InputStream` 。

`org.sonatype.mavenbook.weather.YahooParser`
: `YahooParser` 解析来自 Yahoo! Weather 的 XML，返回 `Weather` 对象。

`org.sonatype.mavenbook.weather.WeatherFormatter`
: `WeatherFormatter` 接受 `Weather` 对象，创建 `VelocityContext` ，根据 Velocity 模板生成结果。

这里我们不是想要详细阐述样例中的代码，但解释一下程序中使之运行的核心代码还是必要的。 我们假设大部分读者已经下载的本书的源码，但也不会忘记那些按着书一步一步往下看的读者。 本小节列出了 `simple-weather` 项目的类，这些类都放在同一个包下面，org.sonatype.mavenbook.weather 。

让我们删掉由 `archetype:create` 生成 `App` 类和 `AppTest` 类，然后加入我们新的包。 在 Maven 项目中，所有项目的源代码都存储在 `src/main/java` 目录。 在新项目的基础目录下，运行下面的命令：

```
$ cd src/test/java/org/sonatype/mavenbook
$ rm AppTest.java
$ cd ../../../../../..
$ cd src/main/java/org/sonatype/mavenbook
$ rm App.java
$ mkdir weather
$ cd weather
```

你已经创建了一个新的包 org.sonatype.mavenbook.weather 。 现在，我们需要把那些类放到这个目录下面。 用你最喜欢的编辑器，创建一个新文件，名字为 `Weather.java`，内容如下：

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

例 4.4. Simple Weather 的 Weather 模型对象

```
package org.sonatype.mavenbook.weather;


public class Weather {
  private String city;
  private String region;
  private String country;
  private String condition;
  private String temp;
  private String chill;
  private String humidity;

  public Weather() {}

  public String getCity() { return city; }
  public void setCity(String city) { this.city = city; }

  public String getRegion() { return region; }
  public void setRegion(String region) { this.region = region; }

  public String getCountry() { return country; }
  public void setCountry(String country) { this.country = country; }

  public String getCondition() { return condition; }
  public void setCondition(String condition) { this.condition = condition; }

  public String getTemp() { return temp; }
  public void setTemp(String temp) { this.temp = temp; }

  public String getChill() { return chill; }
  public void setChill(String chill) { this.chill = chill; }

  public String getHumidity() { return humidity; }
  public void setHumidity(String humidity) { this.humidity = humidity; }
}
```

`Weather` 类定义了一个简单的 bean ，用来存储由 Yahoo! Weather 数据源解析出来的天气信息。天气数据源提供了丰富的信息，从日出日落时间，到风速和风向。 为了让这个例子保持简单， `Weather` 模型对象只保存温度，湿度和当前天气情况的文字描述等信息。

在同一目录下，创建 `Main.java` 文件。`Main` 这个类有一个静态的 `main()` 函数——样例程序的入口。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

例 4.5. Simple Weather 的 Main 类

```
package org.sonatype.mavenbook.weather;

import java.io.InputStream;

import org.apache.log4j.PropertyConfigurator;


public class Main {

  public static void main(String[] args) throws Exception {
    // Configure Log4J
    PropertyConfigurator.configure(Main.class.getClassLoader()
                                       .getResource("log4j.properties"));

    // Read the Zip Code from the Command-line (if none supplied, use 60202)
    int zipcode = 60202;
    try {
      zipcode = Integer.parseInt(args[0]);
    } catch( Exception e ) {}

    // Start the program
    new Main(zipcode).start();
  }

  private int zip;

  public Main(int zip) {
    this.zip = zip;
  }

  public void start() throws Exception {
    // Retrieve Data
    InputStream dataIn = new YahooRetriever().retrieve( zip );

    // Parse Data
    Weather weather = new YahooParser().parse( dataIn );

    // Format (Print) Data
    System.out.print( new WeatherFormatter().format( weather ) );
  }
}
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

上例中的 main() 函数通过获取 classpath 中的资源文件来配置 Log4J ，之后它试图从命令行读取邮政编码。 如果在读取邮政编码的时候抛出了异常，程序会设置默认邮政编码为 60202 。 一旦有了邮政编码，它初始化一个 Main 对象，调用该对象的 start() 方法。而 start() 方法会调用 YahooRetriever 来获取天气的 XML 数据。YahooRetriever 返回一个 InputStreem ，传给 YahooParser 。 YahooParser 解析 XML 数据并返回 Weather 对象。 最后，WeatherFormatter 接受一个 Weather 对象并返回一个格式化的 String ，打印到标准输出。

在相同目录下创建文件 YahooRetriever.java ，内容如下：

例 4.6. Simple Weather 的 YahooRetriever 类

```
package org.sonatype.mavenbook.weather;

import java.io.InputStream;
import java.net.URL;
import java.net.URLConnection;

import org.apache.log4j.Logger;

public class YahooRetriever {

  private static Logger log = Logger.getLogger(YahooRetriever.class);

  public InputStream retrieve(int zipcode) throws Exception {
    log.info( "Retrieving Weather Data" );
    String url = "http://weather.yahooapis.com/forecastrss?p=" + zipcode;
    URLConnection conn = new URL(url).openConnection();
    return conn.getInputStream();
  }
}
```

这个简单的类打开一个连接到 Yahoo! Weather API 的 URLConnection 并返回一个 InputStream 。 我们还需要在该目录下创建文件 YahooParser.java 用以解析这个数据源。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛


```
package org.sonatype.mavenbook.weather;

import java.io.InputStream;
import java.util.HashMap;
import java.util.Map;

import org.apache.log4j.Logger;
import org.dom4j.Document;
import org.dom4j.DocumentFactory;
import org.dom4j.io.SAXReader;

public class YahooParser {

  private static Logger log = Logger.getLogger(YahooParser.class);

  public Weather parse(InputStream inputStream) throws Exception {
    Weather weather = new Weather();

    log.info( "Creating XML Reader" );
    SAXReader xmlReader = createXmlReader();
    Document doc = xmlReader.read( inputStream );

    log.info( "Parsing XML Response" );
    weather.setCity( doc.valueOf("/rss/channel/y:location/@city") );
    weather.setRegion( doc.valueOf("/rss/channel/y:location/@region") );
    weather.setCountry( doc.valueOf("/rss/channel/y:location/@country") );
    weather.setCondition( doc.valueOf("/rss/channel/item/y:condition/@text") );
    weather.setTemp( doc.valueOf("/rss/channel/item/y:condition/@temp") );
    weather.setChill( doc.valueOf("/rss/channel/y:wind/@chill") );
    weather.setHumidity( doc.valueOf("/rss/channel/y:atmosphere/@humidity") );

    return weather;
  }

  private SAXReader createXmlReader() {
    Map<String,String> uris = new HashMap<String,String>();
        uris.put( "y", "http://xml.weather.yahoo.com/ns/rss/1.0" );

    DocumentFactory factory = new DocumentFactory();
    factory.setXPathNamespaceURIs( uris );

    SAXReader xmlReader = new SAXReader();
    xmlReader.setDocumentFactory( factory );
    return xmlReader;
  }
}
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

YahooParser 是本例中最复杂的类，我们不会深入 Dom4J 或者 Jaxen 的细节，但是这个类还是需要一些解释。YahooParser 的 parse() 方法接受一个 InputStrem 然后返回一个 Weather 对象。 为了完成这一目标，它需要用 Dom4J 来解析 XML 文档。因为我们对 Yahoo! Weather XML 命名空间的元素感兴趣，我们需要用 createXmlReader() 方法创建一个包含命名空间信息的 SAXReader 。 一旦我们创建了这个 reader 并且解析了文档，得到了返回的 org.dom4j.Document ，只需要简单的使用 XPath 表达式来获取需要的信息，而不是遍历所有的子元素。 本例中 Dom4J 提供了 XML 解析功能，而 Jaxen 提供了 XPath 功能。

我们已经创建了 Weather 对象，我们需要格式化输出以供人阅读。 在同一目录中创建一个名为 WeatherFormatter.java 的文件。

例 4.8. Simple Weather 的 WeatherFormatter 类

```
package org.sonatype.mavenbook.weather;

import java.io.InputStreamReader;
import java.io.Reader;
import java.io.StringWriter;

import org.apache.log4j.Logger;
import org.apache.velocity.VelocityContext;
import org.apache.velocity.app.Velocity;

public class WeatherFormatter {

  private static Logger log = Logger.getLogger(WeatherFormatter.class);

  public String format( Weather weather ) throws Exception {
    log.info( "Formatting Weather Data" );
    Reader reader =
      new InputStreamReader( getClass().getClassLoader()
                                 .getResourceAsStream("output.vm"));
    VelocityContext context = new VelocityContext();
    context.put("weather", weather );
    StringWriter writer = new StringWriter();
    Velocity.evaluate(context, writer, "", reader);
    return writer.toString();
  }
}
```

WeatherFormatter 使用 VelotNicy 来呈现一个模板。format() 方法接受一个 Weather bean 然后返回格式化好的 String 。format() 方法做的第一件事是从 classpath 载入名字为 output.vm 的 Velocity 模板。 然后我们创建一个 VelocityContext ，它

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

需要一个 Weather 对象来填充。 一个StringWriter被创建用来存放模板生成的结果数据。通过调用 Velocity.evaluate() ，给模板赋值，结果作为 String 返回。

在我们能够运行该样例程序之前，我们需要往 classpath 添加一些资源。

4.7. 添加资源

本项目依赖于两个 classpath 资源： Main 类通过 classpath 资源 log4j.preoperties 来配置 Log4J ， WeatherFormatter 引用了一个在 classpath 中的名为 output.vm 的 Velocity 模板。这两个资源都需要在默认包中（或者 classpath 的根目录）。

为了添加这些资源，我们需要在项目的基础目录下创建一个新的目录—— src/main/resources。 由于任务 archetype:create 没有创建这个目录，我们需要通过在项目的基础目录下运行下面的命令来创建它：

```
$ cd src/main
$ mkdir resources
$ cd resources
```

在这个资源目录创建好之后，我们可以加入这两个资源。首先，往目录 resources 加入文件 log4j.properties。

例 4.9. Simple Weather 的 Log4J 配置文件

```
# Set root category priority to INFO and its only appender to CONSOLE.
log4j.rootCategory=INFO, CONSOLE

# CONSOLE is set to be a ConsoleAppender using a PatternLayout.
log4j.appender.CONSOLE=org.apache.log4j.ConsoleAppender
log4j.appender.CONSOLE.Threshold=INFO
log4j.appender.CONSOLE.layout=org.apache.log4j.PatternLayout
log4j.appender.CONSOLE.layout.ConversionPattern=%-4r %-5p %c{1} %x - %m%n
```

这个 log4j.properties 文件简单配置了 Log4J ，使其使用 PatternLayout 往标准输出打印所有日志信息。 最后，我们需要创建 output.vm ，它是这个命令行程序用来呈现输出的 Velocity 模板。 在 resources 目录创建 output.vm 。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

例 4.10. Simple Weather 的 Output Velocity 模板

```
*********************************
 Current Weather Conditions for:
  ${weather.city}, ${weather.region}, ${weather.country}

 Temperature: ${weather.temp}
   Condition: ${weather.condition}
    Humidity: ${weather.humidity}
  Wind Chill: ${weather.chill}
*********************************
```

这个模板包含了许多对名为 `weather` 的变量的引用。 这个 `weather` 变量是传给 `WeatherFormatter` 的 那个 `Weather` bean， `${weather.temp}` 语法简化的表示获取并显示 `temp` 这个bean属性的值。 现在我们已经在正确的地方有了我们项目的所有代码，我们可以使用 Maven 来运行这个样例。

4.8. 运行Simple Weather项目

使用来自 Codehaus Mojo 项目[2] 的 Exec 插件，我们可以运行这个程序。在项目的基础目录下运行以下命令，以运行该程序的 `Main` 类。

```
$ mvn install
$ mvn exec:java -Dexec.mainClass=org.sonatype.mavenbook.weather.Main
...
[INFO] [exec:java]
0    INFO  YahooRetriever  - Retrieving Weather Data
134  INFO  YahooParser  - Creating XML Reader
333  INFO  YahooParser  - Parsing XML Response
420  INFO  WeatherFormatter  - Formatting Weather Data
*********************************
 Current Weather Conditions for:
  Evanston, IL, US

 Temperature: 45
   Condition: Cloudy
    Humidity: 76
  Wind Chill: 38
*********************************
...
```

我们没有为 `Main` 类提供命令行参数，因此程序按照默认的邮编执行——60202。 正如你能看到的，我们已经成功的运行了 SImple Weather 命令行工具，从 Yahoo!

[2] http://mojo.codehaus.org

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

Weather 获取了一些数据，解析了结果，并且通过 Velocity 格式化了结果数据。 我们仅仅写了项目的源代码，往 pom.xml 添加了一些最少的配置。 注意我们这里没有引入“构建过程”。 我们不需要定义如何或哪里让 Java 编译器编译我们的源代码，我们不需要指导构建系统在运行样例程序的时候如何定位二进制文件， 我们所需要做的是包含一些依赖，用来定位合适的 Maven 坐标。

4.8.1. Maven Exec 插件

Exec 插件允许你运行 Java 类和其它脚本。 它不是 Maven 核心插件，但它可以从 Codehaus[3] 的 Mojo[4] 项目得到。想要查看 Exec 插件的完整描述，运行：

```
$ mvn help:describe -Dplugin=exec -Dfull
```

这会列出所有 Maven Exec 插件可用的目标。 Help 插件同时也会列出 Exec 插件的有效参数，如果你想要定制 Exec 插件的行为，传入命令行参数，你应该使用 help:describe 提供的文档作为指南。 虽然 Exec 插件很有用，在开发过程中用来运行测试之外，你不应该依赖它来运行你的应用程序。 想要更健壮的解决方案，使用 Maven Assembly 插件，它在第 4.13 节 “构建一个打包好的命令行应用程序”中被描述。

4.8.2. 浏览你的项目依赖

Exec 插件让我们能够在不往 classpath 载入适当的依赖的情况下，运行这个程序。 在任何其它的构建系统能够中，我们必须复制所有程序依赖到类似于 lib/ 的目录，这个目录包含一个 JAR 文件的集合。 那样，我们就必须写一个简单的脚本，在 classpath 中包含我们程序的二进制代码和我们的依赖。 只有那样我们才能运行 java org.sonatype.mavenbook.weather.Main 。 Exec 能做这样的工作是因为 Maven 已经知道如何创建和管理你的 classpath 和你的依赖。

了解你项目的 classpath 包含了哪些依赖是很方便也很有用的。这个项目不仅包含了一些类库如 Dom4J， Log4J， Jaxen， 和 Velocity，它同时也引入了一些传递性依赖。 如果你需要找出 classpath 中有什么，你可以使用 Maven Dependency 插件来打印出已解决依赖的列表。 要打印出 Simple Weather 项目的这个列表，运行 dependency:resolve 目标。

```
$ mvn dependency:resolve
...
[INFO] [dependency:resolve]
[INFO]
[INFO] The following files have been resolved:
[INFO]    com.ibm.icu:icu4j:jar:2.6.1 (scope = compile)
[INFO]    commons-collections:commons-collections:jar:3.1 (scope = compile)
```

[3] http://www.codehaus.org
[4] http://mojo.codehaus.org

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛


```
[INFO]    commons-lang:commons-lang:jar:2.1 (scope = compile)
[INFO]    dom4j:dom4j:jar:1.6.1 (scope = compile)
[INFO]    jaxen:jaxen:jar:1.1.1 (scope = compile)
[INFO]    jdom:jdom:jar:1.0 (scope = compile)
[INFO]    junit:junit:jar:3.8.1 (scope = test)
[INFO]    log4j:log4j:jar:1.2.14 (scope = compile)
[INFO]    oro:oro:jar:2.0.8 (scope = compile)
[INFO]    velocity:velocity:jar:1.5 (scope = compile)
[INFO]    xalan:xalan:jar:2.6.0 (scope = compile)
[INFO]    xerces:xercesImpl:jar:2.6.2 (scope = compile)
[INFO]    xerces:xmlParserAPIs:jar:2.6.2 (scope = compile)
[INFO]    xml-apis:xml-apis:jar:1.0.b2 (scope = compile)
[INFO]    xom:xom:jar:1.0 (scope = compile)
```

正如你能看到的，我们项目拥有一个很大的依赖集合。 虽然我们只是为四个类库引入了直接的依赖，看来我们实际共引入了15个依赖。 Dom4J 依赖于 Xerces 和 XML 解析器 API ，Jaxen 依赖于 Xalan，后者也就在 classpath 中可用了。 Dependency 插件将会打印出最终的你项目编译所基于的所有依赖的组合。 如果你想知道你项目的整个依赖树，你可以运行 dependency:tree 目标。

```
$ mvn dependency:tree
...
[INFO] [dependency:tree]
[INFO] org.sonatype.mavenbook.ch04:simple-weather:jar:1.0
[INFO] +- log4j:log4j:jar:1.2.14:compile
[INFO] +- dom4j:dom4j:jar:1.6.1:compile
[INFO] |  \- xml-apis:xml-apis:jar:1.0.b2:compile
[INFO] +- jaxen:jaxen:jar:1.1.1:compile
[INFO] |  +- jdom:jdom:jar:1.0:compile
[INFO] |  +- xerces:xercesImpl:jar:2.6.2:compile
[INFO] |  \- xom:xom:jar:1.0:compile
[INFO] |     +- xerces:xmlParserAPIs:jar:2.6.2:compile
[INFO] |     +- xalan:xalan:jar:2.6.0:compile
[INFO] |     \- com.ibm.icu:icu4j:jar:2.6.1:compile
[INFO] +- velocity:velocity:jar:1.5:compile
[INFO] |  +- commons-collections:commons-collections:jar:3.1:compile
[INFO] |  +- commons-lang:commons-lang:jar:2.1:compile
[INFO] |  \- oro:oro:jar:2.0.8:compile
[INFO] +- org.apache.commons:commons-io:jar:1.3.2:test
[INFO] \- junit:junit:jar:3.8.1:test
...
```

如果你还不满足，或者想要查看完整的依赖踪迹，包含那些因为冲突或者其它原因而被拒绝引入的构件，打开 Maven 的调试标记运行：

```
$ mvn install -X
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛


```
...
[DEBUG] org.sonatype.mavenbook.ch04:simple-weather:jar:1.0 (selected for null)
[DEBUG]   log4j:log4j:jar:1.2.14:compile (selected for compile)
[DEBUG]   dom4j:dom4j:jar:1.6.1:compile (selected for compile)
[DEBUG]     xml-apis:xml-apis:jar:1.0.b2:compile (selected for compile)
[DEBUG]   jaxen:jaxen:jar:1.1.1:compile (selected for compile)
[DEBUG]     jaxen:jaxen:jar:1.1-beta-6:compile (removed - causes a cycle in the gra
[DEBUG]     jaxen:jaxen:jar:1.0-FCS:compile (removed - causes a cycle in the graph)
[DEBUG]     jdom:jdom:jar:1.0:compile (selected for compile)
[DEBUG]     xml-apis:xml-apis:jar:1.3.02:compile (removed - nearer found: 1.0.b2)
[DEBUG]     xerces:xercesImpl:jar:2.6.2:compile (selected for compile)
[DEBUG]     xom:xom:jar:1.0:compile (selected for compile)
[DEBUG]       xerces:xmlParserAPIs:jar:2.6.2:compile (selected for compile)
[DEBUG]       xalan:xalan:jar:2.6.0:compile (selected for compile)
[DEBUG]       xml-apis:xml-apis:1.0.b2.
[DEBUG]       com.ibm.icu:icu4j:jar:2.6.1:compile (selected for compile)
[DEBUG]   velocity:velocity:jar:1.5:compile (selected for compile)
[DEBUG]     commons-collections:commons-collections:jar:3.1:compile (selected for co
[DEBUG]     commons-lang:commons-lang:jar:2.1:compile (selected for compile)
[DEBUG]     oro:oro:jar:2.0.8:compile (selected for compile)
[DEBUG]   junit:junit:jar:3.8.1:test (selected for test)
```

从调试输出我们看到一些依赖管理系统工作的内部信息。 你在这里看到的是项目的依赖树。 Maven 正打印出你项目的所有的依赖，以及这些依赖的依赖（还有依赖的依赖的依赖）的完整的 Maven 坐标。 你能看到 `simple-weather` 依赖于 `jaxen` ， `jaxen` 依赖于 `xom` ， `xom` 接着依赖于 `icu4j` 。从该输出你能看到 Maven 正在创建一个依赖图，排除重复，解决不同版本之间的冲突。 如果你的依赖有问题，通常在 `dependency:tree` 所生成的列表基础上更深入一点会有帮助；开启调试输出允许你看到 Maven 工作时的依赖机制。

4.9. 编写单元测试

Maven 内建了对单元测试的支持，测试是 Maven 默认生命周期的一部分。让我们给 Simple Weather 项目添加一些单元测试。 首先，在 `src/test/java` 下面创建包 org.sonatype.mavenbook.weather。

```
$ cd src/test/java
$ cd org/sonatype/mavenbook
$ mkdir -p weather/yahoo
$ cd weather/yahoo
```

目前，我们将会创建两个单元测试。 第一个单元测试会测试 `YahooParser` ，第二个会测试 `WeatherFormatter`。 在 weather 包中，创建一个带有一以下内容的文件，名称为 `YahooParserTest.java` 。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

例 4.11. Simple Weather 的 YahooParserTest 单元测试

```
package org.sonatype.mavenbook.weather.yahoo;

import java.io.InputStream;

import junit.framework.TestCase;

import org.sonatype.mavenbook.weather.Weather;
import org.sonatype.mavenbook.weather.YahooParser;

public class YahooParserTest extends TestCase {

  public YahooParserTest(String name) {
    super(name);
  }

  public void testParser() throws Exception {
    InputStream nyData =
      getClass().getClassLoader().getResourceAsStream("ny-weather.xml");
    Weather weather = new YahooParser().parse( nyData );
    assertEquals( "New York", weather.getCity() );
    assertEquals( "NY", weather.getRegion() );
    assertEquals( "US", weather.getCountry() );
    assertEquals( "39", weather.getTemp() );
    assertEquals( "Fair", weather.getCondition() );
    assertEquals( "39", weather.getChill() );
    assertEquals( "67", weather.getHumidity() );
  }
}
```

`YahooParserTest` 继承了 JUnit 定义的 `TestCase` 类。 它遵循了 JUnit 测试的惯例模式：一个构造函数接受一个单独的 `String` 参数并调用父类的构造函数，还有一系列以“test”开头的公有方法，做为单元测试被调用。 我们定义了一个单独的测试方法， `testParser` ，通过解析一个值已知的 XML 文档来测试 `YahooParser` 。 测试 XML 文档命名为 `ny-weather.xml` ，从 classpath 载入。我们将在第 4.11 节 “添加单元测试资源”添加测试资源。 在我们这个 Maven 项目的目录布局中，文件 `ny-weather.xml` 可以从包含测试资源的目录—— `/usr/local/hudson/hudson-home/jobs/maven-guide-zh-to-production/workspace/content-zh/src/test/resources` ——中找到，路径为 `org/sonatype/mavenbook/weather/yahoo/ny-weather.xml` 。 该文件作为一个 `InputStream` 被读入，传给 `YahooParser` 的 `parse()` 方法。 `parse()` 方法返回一个 `Weather` 对象，该对象通过一系列 由 `TestCase` 定义的 `assertEquals()` 调用而被测试。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

在同一目录下创建一个名为 WeatherFormatterTest.java 的文件。

例 4.12. Simple Weather 的 WeatherFormatterTest 单元测试

```
package org.sonatype.mavenbook.weather.yahoo;

import java.io.InputStream;

import org.apache.commons.io.IOUtils;

import org.sonatype.mavenbook.weather.Weather;
import org.sonatype.mavenbook.weather.WeatherFormatter;
import org.sonatype.mavenbook.weather.YahooParser;

import junit.framework.TestCase;

public class WeatherFormatterTest extends TestCase {

  public WeatherFormatterTest(String name) {
    super(name);
  }

  public void testFormat() throws Exception {
    InputStream nyData =
      getClass().getClassLoader().getResourceAsStream("ny-weather.xml");
    Weather weather = new YahooParser().parse( nyData );
    String formattedResult = new WeatherFormatter().format( weather );
    InputStream expected =
      getClass().getClassLoader().getResourceAsStream("format-expected.dat");
    assertEquals( IOUtils.toString( expected ).trim(), formattedResult.trim() );
  }
}
```

该项目中的第二个单元测试测试 WeatherFormatter。 和 YahooParserTest 一样，WeatherFormatter 同样也继承 JUnit 的 TestCase 类。 这个单独的测试通过单元测试的 classpath 从 /usr/local/hudson/hudson-home/jobs/maven-guide-zh-to-production/workspace/content-zh/src/test/resources 的 org/sonatype/mavenbook/weather/yahoo 目录读取同样的测试资源文件。 我们将会在第 4.11 节 “添加单元测试资源”添加测试资源。 WeatherFormatterTest 首先调用 YahooParser 解析出 Weather 对象，然后用 WeatherFormatter 格式化这个对象。 我们的期望输出被存储在一个名为 format-expected.dat 的文件中，该文件存放在和 ny-weather.xml 同样的目录中。 要比较测试输出和期望输出，我们将期望输出作为 InputStream 读入，然后使用 Commons IO 的 IOUtils 类来把文件转化为 String 。 然后使用 assertEquals() 比较这个 String 和测试输出。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

4.10. 添加测试范围依赖

在类 WeatherFormatterTest 中我们用了一个来自于 Apache Commons IO 的工具——IOUtils 类。 IOUtils 提供了许多很有帮助的静态方法，能帮助让很多工作摆脱繁琐的 I/O 操作。在这个单元测试中我们使用了 IOUtils.toString() 来复制 classpath 中资源 format.expected.dat 中的数据至 String。 不用 Commons IO 我们也能完成这件事情，但是那需要额外的六七行代码来处理像 InputStreamReader 和 StringWriter 这样的对象。我们使用 Commons IO 的主要原因是，能有理由添加对 Commons IO 的测试范围依赖。

测试范围依赖是一个只在测试编译和测试运行时在 classpath 中有效的依赖。如果你的项目是以 war 或者 ear 形式打包的，测试范围依赖就不会被包含在项目的打包输出中。 要添加一个测试范围依赖，在你项目的 dependencies 小节中添加如下 dependency 元素。

例 4.13. 添加一个测试范围依赖

```
<project>
  ...
  <dependencies>
    ...
    <dependency>
      <groupId>org.apache.commons</groupId>
      <artifactId>commons-io</artifactId>
      <version>1.3.2</version>
      <scope>test</scope>
    </dependency>
    ...
  </dependencies>
</project>
```

当你往 pom.xml 中添加了这个依赖以后，运行 mvn dependency:resolve 你会看到 commons-io 出现在在依赖列表中，范围是 test 。 在我们可以运行该项目的单元测试之前，我们还需要做一件事情。 那就是创建单元测试依赖的 classpath 资源。测试范围依赖将在 9.4.1节 “依赖范围” 中详细解释。

4.11. 添加单元测试资源

一个单元测试需要访问针对测试的一组资源。 通常你需要在测试 classpath 中存储一些包含期望结果的文件，以及包含模拟输入的文件。 在本项目中，我们为 YahooParserTest 准备了一个名为 ny-weather.xml 的测试 XML 文档，还有一个名为 format-expected.dat 的文件，包含了 WeatherFormatter 的期望输出。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

要添加测试资源，你需要创建目录 `src/test/resources` 。 这是 Maven 寻找测试资源的默认目录。 在你的项目基础目录下运行下面的命令以创建该目录。

```
$ cd src/test
$ mkdir resources
$ cd resources
```

当你创建好这个资源目录之后，在资源目录下创建一个名为 `format-expected.dat` 的文件。

例 4.14. Simple Weather 的 WeatherFormatterTest 期望输出

```
*********************************
 Current Weather Conditions for:
  New York, NY, US

 Temperature: 39
   Condition: Fair
    Humidity: 67
  Wind Chill: 39
*********************************
```

这个文件应该看起来很熟悉了，它和你用 Maven Exec 插件运行 Simple Weather 项目得到的输出是一样的。你需要在资源目录添加的第二个文件是 `ny-weather.xml` 。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

```
     xmlns:geo="http://www.w3.org/2003/01/geo/wgs84_pos#">
 <title>Yahoo! Weather - New York, NY</title>
 <link>http://us.rd.yahoo.com/dailynews/rss/weather/New_York__NY/</link>
 <description>Yahoo! Weather for New York, NY</description>
 <language>en-us</language>
 <lastBuildDate>Sat, 10 Nov 2007 8:51 pm EDT</lastBuildDate>

 <ttl>60</ttl>
 <yweather:location city="New York" region="NY" country="US" />
 <yweather:units temperature="F" distance="mi" pressure="in" speed="mph" />
 <yweather:wind chill="39" direction="0" speed="0" />
 <yweather:atmosphere humidity="67" visibility="1609" pressure="30.18"
                      rising="1" />
  <yweather:astronomy sunrise="6:36 am" sunset="4:43 pm" />
  <image>
 <title>Yahoo! Weather</title>

 <width>142</width>
 <height>18</height>
 <link>http://weather.yahoo.com/</link>
 <url>http://l.yimg.com/us.yimg.com/i/us/nws/th/main_142b.gif</url>
 </image>
 <item>
 <title>Conditions for New York, NY at 8:51 pm EDT</title>

  <geo:lat>40.67</geo:lat>
 <geo:long>-73.94</geo:long>
  <link>http://us.rd.yahoo.com/dailynews/rss/weather/New_York__NY/\</link>
 <pubDate>Sat, 10 Nov 2007 8:51 pm EDT</pubDate>
 <yweather:condition text="Fair" code="33" temp="39"
                     date="Sat, 10 Nov 2007 8:51 pm EDT" />
 <description><![CDATA[
<img src="http://l.yimg.com/us.yimg.com/i/us/we/52/33.gif" /><br />
 <b>Current Conditions:</b><br />
 Fair, 39 F<BR /><BR />
 <b>Forecast:</b><BR />
  Sat - Partly Cloudy. High: 45 Low: 32<br />
  Sun - Sunny. High: 50 Low: 38<br />
 <br />
 ]]></description>
 <yweather:forecast day="Sat" date="10 Nov 2007" low="32" high="45"
                   text="Partly Cloudy" code="29" />

<yweather:forecast day="Sun" date="11 Nov 2007" low="38" high="50"
                   text="Sunny" code="32" />
  <guid isPermaLink="false">10002_2007_11_10_20_51_EDT</guid>
 </item>
</channel>
</rss>
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

该文件包含了一个给 YahooParserTest 用的 XML 文档。有了这个文件，我们不用从 Yahoo! Weather 获取 XML 响应就能测试 YahooParser 了。

4.12. 执行单元测试

既然你的项目已经有单元测试了，那么让它们运行起来吧。 你不必为了运行单元测试做什么特殊的事情， test 阶段是 Maven 生命周期中常规的一部分。 当你运行 mvn package 或者 mvn install 的时候你也运行了测试。 如果你想要运行到 test 阶段为止的所有生命周期阶段，运行 mvn test 。

```
$ mvn test
...
[INFO] [surefire:test]
[INFO] Surefire report directory: ~/examples/simple-weather/target/surefire-reports

-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running org.sonatype.mavenbook.weather.yahoo.WeatherFormatterTest
0    INFO  YahooParser  - Creating XML Reader
177  INFO  YahooParser  - Parsing XML Response
239  INFO  WeatherFormatter  - Formatting Weather Data
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.547 sec
Running org.sonatype.mavenbook.weather.yahoo.YahooParserTest
475  INFO  YahooParser  - Creating XML Reader
483  INFO  YahooParser  - Parsing XML Response
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.018 sec

Results :

Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
```

从命令行运行 mvn test 使 Maven 执行到 test 阶段为止的所有生命周期阶段。 Maven Surefire 插件有一个 test 目标，该目标被绑定在了 test 阶段。 test 目标执行项目中所有能在 src/test/java 找到的并且文件名与 **/Test*.java， **/*Test.java 和 **/*TestCase.java 匹配的所有单元测试。 在本例中，你能看到 Surefire 插件的 test 目标执行了 WeatherFormatterTest 和 YahooParserTest 。 在 Maven Surefire 插件执行 JUnit 测试的时候，它同时也在 /usr/local/hudson/hudson-home/jobs/maven-guide-zh-to-production/workspace/content-zh/target/surefire-reports 目录下生成 XML 和常规文本报告。 如果你的测试失败了，你可以去查看这个目录，里面有你单元测试生成的异常堆栈信息和错误信息。

4.12.1. 忽略测试失败

通常，你会开发一个带有很多失败单元测试的系统。 如果你正在实践测试驱动开发(TDD)，你可能会使用测试失败来衡量你离项目完成有多远。 如果你有失败的单元测试，但你仍然希望产生构建输出，你就必须告诉 Maven 让它忽略测试失败。 当 Maven 遇到一个测试失败，它默认的行为是停止当前的构建。 如果你希望继续构建项目，即使 Surefire 插件遇到了失败的单元测试，你就需要设置 Surefire 的 testFailureIgnore 这个配置属性为 true。

例 4.16. 忽略单元测试失败

```
<project>
  [...]
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-surefire-plugin</artifactId>
        <configuration>
          <testFailureIgnore>true</testFailureIgnore>
        </configuration>
      </plugin>
    </plugins>
  </build>
  [...]
</project>
```

该插件文档（http://maven.apache.org/plugins/maven-surefire-plugin/test-mojo.html）说明，这个参数声明为一个表达式：

例 4.17. 插件参数表达式

```
       testFailureIgnore  Set this to true to ignore a failure during testing. Its

    * Type: boolean
    * Required: No
    * Expression: ${maven.test.failure.ignore}
```

这个表达式可以从命令行通过 -D 参数设置。

```
$ mvn test -Dmaven.test.failure.ignore=true
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

4.12.2. 跳过单元测试

你可能想要配置 Maven 使其完全跳过单元测试。 可能你有一个很大的系统，单元测试需要花好多分钟来完成，而你不想在生成最终输出前等单元测试完成。 你可能正工作在一个遗留系统上面，这个系统有一系列的失败的单元测试，你可能仅仅想要生成一个 JAR 而不是去修复所有的单元测试。 Maven 提供了跳过单元测试的能力，只需要使用 Surefire 插件的 `skip` 参数。 在命令行，只要简单的给任何目标添加 `maven.test.skip` 属性就能跳过测试：

```
$ mvn install -Dmaven.test.skip=true
...
[INFO] [compiler:testCompile]
[INFO] Not compiling test sources
[INFO] [surefire:test]
[INFO] Tests are skipped.
...
```

当 Surefire 插件到达 `test` 目标的时候，如果 `maven.test.skip` 设置为 `true` ，它就会跳过单元测试。 另一种配置 Maven 跳过单元测试的方法是给你项目的 `pom.xml` 添加这个配置。 你需要为你的 `build` 添加 `plugin` 元素。

例 4.18. 跳过单元测试

```
<project>
  [...]
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-surefire-plugin</artifactId>
        <configuration>
          <skip>true</skip>
        </configuration>
      </plugin>
    </plugins>
  </build>
  [...]
</project>
```

4.13. 构建一个打包好的命令行应用程序

在 第 4.12 节 “执行单元测试”我们使用 Maven Exec 插件运行了 Simple Weather 应用程序。虽然 Maven Exec 能运行程序并且产生输出，你不能就把 Maven 当成是你程序运行的容器。 如果你把这个命令行程序分发给其他人，你大概就需要

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

分发一个 JAR 或者一个 ZIP 存档文件或者 TAR 压缩过的 GZIP 文件。下面的小节介绍了使用 Maven Assembly 插件的预定义装配描述符生成一个可分发的 JAR 文件的过程，该文件包含了项目的二进制文件和所有的依赖。

Maven Assembly 插件是一个用来创建你应用程序特有分发包的插件。 你可以使用 Maven Assembly 插件以你希望的任何形式来装配输出，只需定义一个自定义的装配描述符。 后面的章节我们会说明如何创建一个自定义装配描述符，为 Simple Weather 应用程序生成一个更复杂的存档文件。 本章我们将会使用预定义的 jar-with-dependencies 格式。 要配置 Maven Assembly 插件， 我们需要在 pom.xml 中的 build 配置中添加如下的 plugin 配置。

例 4.19. 配置 Maven 装配描述符

```
<project>
  [...]
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-assembly-plugin</artifactId>
        <configuration>
          <descriptorRefs>
            <descriptorRef>jar-with-dependencies</descriptorRef>
          </descriptorRefs>
        </configuration>
      </plugin>
    </plugins>
  </build>
  [...]
</project>
```

添加好这些配置以后，你可以通过运行 mvn assembly:assembly 来构建这个装配。

```
$ mvn install assembly:assembly
...
[INFO] [jar:jar]
[INFO] Building jar: ~/examples/simple-weather/target/simple-weather-1.0.jar
[INFO] [assembly:assembly]
[INFO] Processing DependencySet (output=)
[INFO] Expanding: \
       .m2/repository/dom4j/dom4j/1.6.1/dom4j-1.6.1.jar into \
       /tmp/archived-file-set.1437961776.tmp
[INFO] Expanding: .m2/repository/commons-lang/commons-lang/2.1/commons-lang-2.1.jar
       into /tmp/archived-file-set.305257225.tmp
... (Maven Expands all dependencies into a temporary directory) ...
[INFO] Building jar: \
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛


```
       ~/examples/simple-weather/target/simple-weather-1.0-jar-with-dependencies.ja
```

在 target/simple-weather-1.0-jar-with-dependencies.jar 装配好之后， 我们可以在命令行重新运行 Main 类。在你项目的基础目录下运行以下命令：

```
$ cd target
$ java -cp simple-weather-1.0-jar-with-dependencies.jar org.sonatype.mavenbook.weat
0    INFO  YahooRetriever  - Retrieving Weather Data
221  INFO  YahooParser  - Creating XML Reader
399  INFO  YahooParser  - Parsing XML Response
474  INFO  WeatherFormatter  - Formatting Weather Data
*********************************
 Current Weather Conditions for:
  New York, NY, US

 Temperature: 44
   Condition: Fair
    Humidity: 40
  Wind Chill: 40
*********************************
```

jar-with-dependencies 格式创建一个包含所有 simple-weather 项目的二进制代码以及所有依赖解压出来的二进制代码的 JAR 文件。 这个略微非常规的格式产生了一个 9 MiB 大小的 JAR 文件，包含了大概 5290 个类。 但是它确实给那些使用 Maven 开发的应用程序提供了一个易于分发的格式。本书的后面，我们会说明如何创建一个自定义的装配描述符来生成一个更标准的分发包。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

第 5 章 一个简单的Web应用

5.1. 介绍

本章我们使用 Maven Archetype 插件创建一个简单的 web 应用程序。 我们将会在一个名为 Jetty 的 Servlet 容器中运行这个 web 应用程序，同时添加一些依赖，编写一个简单的 Servlet，并且生成一个 WAR 文件。 本章最后，你将能够开始使用 Maven 来提高你开发 web 应用程序的速度。

5.1.1. 下载本章样例

本章的样例是通过 Maven Archetype 插件生成的。 虽然没有样例源码你也应该能够理解这个开发过程，但还是推荐你下载样例源码作为参考。 本章的样例项目包含在本书的样例代码中，你可以从两个地方下载，http://www.sonatype.com/book/mvn-examples-1.0.zip 或者 http://www.sonatype.com/book/mvn-examples-1.0.tar.gz 。解开存档文件至任意目录，然后到 `ch05/` 目录。 在 `ch05/` 目录你会看到一个名为 `simple-webapp/` 的目录，它包含了本章开发出来的 Maven 项目。如果你想要在浏览器里看样例代码，访问 http://www.sonatype.com/book/examples-1.0 ，然后点击 `ch05/` 目录。

5.2. 定义这个简单的Web应用

我们已经有意的使本章关注于一个简单 Web 应用（POWA）—— 一个 servlet 和一个 JSP 页面。 在接下来的二十多页中，我们不会告诉你如何开发你的 Struts 2，Tapesty，Wicket，JSF，或者 Waffle 应用，我们也不会涉及到集成诸如 Plexus，Guice 或者 Spring Framework 之类的 IoC 容器。 本章的目标是展示给你看开发 web 应用的时候 Maven 提供的基本设备，不多，也不少。 本书的后面，我们将会看一下开发两个 web 应用，一个使用了 Hibernate， Velocity 和 Spring Framework，另外一个使用了 Plexus。

5.3. 创建这个简单的Web应用

创建你的 web 应用程序项目，运行 mvn archetype:create ，加上参数 `artifactId` 和 `groupId`。 指定 `archetypeArtifactId` 为 `maven-archetype-webapp`。 如此便创建了恰到好处的目录结构和 Maven POM。

```
~/examples$ mvn archetype:create -DgroupId=org.sonatype.mavenbook.ch05 \
                                                      -DartifactId=simple-wel
                                                      -DpackageName=org.sona
                                                      -DarchetypeArtifactId=
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛


```
[INFO] [archetype:create]
[INFO] ------------------------------------------------------------------------
[INFO] Using following parameters for creating Archetype: maven-archetype-webapp:RE
[INFO] ------------------------------------------------------------------------
[INFO] Parameter: groupId, Value: org.sonatype.mavenbook.ch05
[INFO] Parameter: packageName, Value: org.sonatype.mavenbook
[INFO] Parameter: basedir, Value: ~/examples
[INFO] Parameter: package, Value: org.sonatype.mavenbook
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] Parameter: artifactId, Value: simple-webapp
[INFO] ********************* End of debug info from resources from generated POM **
[INFO] Archetype created in dir: ~/examples/simple-webapp
```

在 Maven Archetype 插件创建好了项目之后，切换目录至 simple-web 后看一下 pom.xml 。 你会看到如下的 XML 文档：

例 5.1. simple-web 项目的初始 POM

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
         http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.sonatype.mavenbook.ch05</groupId>
  <artifactId>simple-webapp</artifactId>
  <packaging>war</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>simple-webapp Maven Webapp</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
  <build>
    <finalName>simple-webapp</finalName>
  </build>
</project>
```

注意 packaging 元素包含的值是 war 。这种打包类型配置让 Maven 以 WAR 文件的形式生成一个 web 应用。一个打包类型为 war 的项目，将会在 target/ 目录创建一个

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

WAR 文件，这个文件的默认名称是 content-zh-0.6-SNAPSHOT.war 。对于这个项目，默认的 WAR 文件是 target/simple-webapp-1.0-SNAPSHOT.war 。在这个 simple-webapp 项目中，我们已经通过在项目的构建配置中加入 finalName 元素来自定义这个生成的 WAR 文件的名称。根据 simple-webapp 的 finalName ，package 阶段生成的 WAR 文件为 target/simple-webapp.war 。

5.4. 配置Jetty插件

在你已经编译，测试并且打包了你的 web 应用之后，你会想要将它部署到一个 servlet 容器中，然后测试一下由 Maven Archetype 插件创建的 index.jsp 。通常情况下，你需要下载 Jetty 或者 Apache Tomcat，解压分发包，复制你的应用程序 WAR 文件至 webapps/ 目录，然后启动你的容器。 现在，实现同样的目的，你不再需要做这些事情。 取而代之的是，你可以使用 Maven Jetty 插件在 Maven 中运行你的 web 应用。为此，你需要在项目的 pom.xml 中配置 Maven Jetty 插件。 在你项目的构建配置中添加如下插件元素：

例 5.2. 配置 Jetty 插件

```
<project>
  [...]
  <build>
    <finalName>simple-webapp</finalName>
    <plugins>
      <plugin>
        <groupId>org.mortbay.jetty</groupId>
        <artifactId>maven-jetty-plugin</artifactId>
      </plugin>
    </plugins>
  </build>
  [...]
</project>
```

在项目的 pom.xml中 配置好 Maven Jetty 插件之后，你就可以调用 Jetty 插件的 Run 目标在 Jetty Servlet 容器中启动你的 web 应用。如下运行 mvn jetty:run ：

```
~/examples$ mvn jetty:run
...
[INFO] [jetty:run]
[INFO] Configuring Jetty for project: simple-webapp Maven Webapp
[INFO] Webapp source directory = \
       /Users/tobrien/svnw/sonatype/examples/simple-webapp/src/main/webapp
[INFO] web.xml file = \
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛


```
       /Users/tobrien/svnw/sonatype/examples/simple-webapp/src/main/webapp/WEB-INF/
[INFO] Classes = /Users/tobrien/svnw/sonatype/examples/simple-webapp/target/classes
2007-11-17 22:11:50.532::INFO:  Logging to STDERR via org.mortbay.log.StdErrLog
[INFO] Context path = /simple-webapp
[INFO] Tmp directory =  determined at runtime
[INFO] Web defaults = org/mortbay/jetty/webapp/webdefault.xml
[INFO] Web overrides =  none
[INFO] Webapp directory = \
       /Users/tobrien/svnw/sonatype/examples/simple-webapp/src/main/webapp
[INFO] Starting jetty 6.1.6rc1 ...
2007-11-17 22:11:50.673::INFO:  jetty-6.1.6rc1
2007-11-17 22:11:50.846::INFO:  No Transaction manager found - if your webapp requi
       please configure one.
2007-11-17 22:11:51.057::INFO:  Started SelectChannelConnector@0.0.0.0:8080
[INFO] Started Jetty Server
```

当 Maven 启动了 Jetty Servlet 容器之后，在浏览器中载入 URL http://localhost:8080/simple-webapp/ 。 Archetype 生成的简单页面 index.jsp 没什么价值；它包含了一个文本为“Hello World!”的二级标题。 Maven 认为 web 应用程序的文档根目录为 src/main/webapp 。这个目录就是存放 index.jsp 的目录 。 index.jsp 的内容为 例 5.3 “src/main/webapp/index.jsp 的内容”：

例 5.3. src/main/webapp/index.jsp 的内容

```
<html>
  <body>
    <h2>Hello World!</h2>
  </body>
</html>
```

在 src/main/webapp/WEB-INF 目录中我们会找到可能是最小的 web 应用程序描述符 web.xml。

例 5.4. src/main/webapp/WEB-INF/web.xml 的内容

```
<!DOCTYPE web-app PUBLIC
 "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
 "http://java.sun.com/dtd/web-app_2_3.dtd" >

<web-app>
  <display-name>Archetype Created Web Application</display-name>
</web-app>
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

5.5. 添加一个简单的Servlet

一个只有一个单独的 JSP 页面而没有任何配置好的 servlet 的 web 应用程序基本是没用的。 让我们为这个应用添加一个简单的 servlet ，同时为 pom.xml 和 web.xml 做些改动以支持这个变化。 首先，我们需要在目录 src/main/java 下创建一个名为 org.sonatype.mavenbook.web 的新的包。

```
$ mkdir -p src/main/java/org/sonatype/mavenbook/web
$ cd src/main/java/org/sonatype/mavenbook/web
```

包创建好之后，切换目录至 src/main/java/org/sonatype/mavenbook/web ，创建一个名为 SimpleServlet.java 的 servlet 类，代码如下：

例 5.5. SimpleServlet 类

```
package org.sonatype.mavenbook.web;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class SimpleServlet extends HttpServlet {
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {

        PrintWriter out = response.getWriter();
        out.println( "SimpleServlet Executed" );
        out.flush();
        out.close();
    }
}
```

我们的 SimpleServlet 仅此而已，一个往响应 Writer 打印一条简单信息的 servlet 。为了把这个 servlet 添加到你的 web 应用，并且使其与请求路径匹配，需要添加如下的 servlet 和 servlet-mapping 元素至你项目的 web.xml 文件。 文件 web.xml 可以在目录 src/main/webapp/WEB-INF 中找到。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

例 5.6. 匹配 Simple Servlet

```
<!DOCTYPE web-app PUBLIC
 "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
 "http://java.sun.com/dtd/web-app_2_3.dtd" >

<web-app>
  <display-name>Archetype Created Web Application</display-name>
  <servlet>
    <servlet-name>simple</servlet-name>
    <servlet-class>org.sonatype.mavenbook.web.SimpleServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>simple</servlet-name>
    <url-pattern>/simple</url-pattern>
  </servlet-mapping>
</web-app>
```

一切文件就绪，准备测试这个 servlet 。 src/main/java 下的类，以及 web.xml 已经被更新了。在启动 Jetty 插件之前，运行 mvn compile 以编译你的项目：

```
~/examples$ mvn compile
...
[INFO] [compiler:compile]
[INFO] Compiling 1 source file to ~/examples/ch05/simple-webapp/target/classes
[INFO] ------------------------------------------------------------------------
[ERROR] BUILD FAILURE
[INFO] ------------------------------------------------------------------------
[INFO] Compilation failure

~/ch05/simple-webapp/src/main/java/org/sonatype/mavenbook/web/SimpleServlet.java:[4
  package javax.servlet does not exist

~/ch05/simple-webapp/src/main/java/org/sonatype/mavenbook/web/SimpleServlet.java:[5
  package javax.servlet.http does not exist

~/ch05/simple-webapp/src/main/java/org/sonatype/mavenbook/web/SimpleServlet.java:[7
  cannot find symbol
  symbol: class HttpServlet
  public class SimpleServlet extends HttpServlet {

~/ch05/simple-webapp/src/main/java/org/sonatype/mavenbook/web/SimpleServlet.java:[8
  cannot find symbol
  symbol  : class HttpServletRequest
  location: class org.sonatype.mavenbook.web.SimpleServlet
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛


```
~/ch05/simple-webapp/src/main/java/org/sonatype/mavenbook/web/SimpleServlet.java:[9
  cannot find symbol
  symbol  : class HttpServletResponse
  location: class org.sonatype.mavenbook.web.SimpleServlet

~/ch05/simple-webapp/src/main/java/org/sonatype/mavenbook/web/SimpleServlet.java:[1
  cannot find symbol
  symbol  : class ServletException
  location: class org.sonatype.mavenbook.web.SimpleServlet
```

编译失败了，因为你的 Maven 项目没有对 Servlet API 的依赖。 在下一节，我们会为你项目的 POM 添加 Servlet API 。

5.6. 添加J2EE依赖

为了编写一个 servlet ，我们需要添加 Servlet API 作为项目依赖。 Servlet 规格说明是一个 JAR 文件，它能从 Sun Microsystems 的站点下载到 http://java.sun.com/products/servlet/download.html 。JAR 文件下载好之后你需要把它安装到位于 `~/.m2/repository` 的 Maven 本地仓库。你必须为所有 Sun Microsystems 维护的 J2EE API 重复同样的过程，包括 JNDI， JDBC， Servlet， JSP， JTA， 以及其它。 如果你不想这么做因为觉得这样太无聊了，其实不只有你这么认为。 幸运的是，有一种更简单的方法来下载所有这些类库并安装到本地仓库 —— Apache Geronimo 的独立的开源实现。

很多年以来，获取 Servlet 规格说明 JAR 文件的唯一方式是从 Sun Microsystems 下载。 你必须到 Sun 的 web 站点，同意并且点击它的许可证协议，这样才能访问 Servlet JAR。这是必须的，因为 Sun 的规格说明 JAR 文件并没有使用一个允许再次分发的许可证。 很多年来编写一个 Servlet 或者使用 JDBC 之前你必须手工下载 Sun 的构件。 这很乏味并且令人恼火，直到 Apache Geronimo 项目创建了很多通过 Sun 认证的企业级规格说明实现。 这些规格说明 JAR 是按照 Apache 软件许可证版本 2.0 发布的，该许可证允许对源代码和二进制文件进行免费的再次分发。 现在，对你的程序来说，从 Sun Microsystems 下载的 Servlet API JAR 和从 Apache Geronimo 项目下载的 JAR 没什么大的差别。 它们同样都通过了 Sun Microsystems 的严格的一个测试兼容性工具箱(TCK)。

添加像 JSP API 或者 Servlet API 这样的依赖现在很简单明了了，不再需要你从 web 站点手工下载 JAR 文件然后再安装到本地仓库。 关键是你必须知道去哪里找，使用什么 `groupId`， `artifactId`， 和 `version` 来引用恰当的 Apache Geronimo 实现。给 `pom.xml` 添加如下的依赖元素以添加对 Servlet 规格说明 API 的依赖。.

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

例 5.7. 添加 Servlet 2.4 规格说明作为依赖

```
<project>
  [...]
  <dependencies>
    [...]
    <dependency>
      <groupId>org.apache.geronimo.specs</groupId>
      <artifactId>geronimo-servlet_2.4_spec</artifactId>
      <version>1.1.1</version>
      <scope>provided</scope>
    </dependency>
  </dependencies>
  [...]
</project>
```

所有 Apache Geronimo 规格说明的实现的 groupId 都是 org.apache.geronimo.specs 。这个 artifactId 包含了你熟悉的规格说明的版本号；例如，如果你要引入 Servlet 2.3 规格说明，你将使用的 artifactId 是 geronimo-servlet_2.3_spec ，如果你想要引入 Servlet 2.4 规格说明，那么你的 artifactId 将会是 geronimo-servlet_2.4_spec 。你必须看一下 Maven 的公共仓库来决定使用什么版本。 最好使用某个规格说明实现的最新版本。 如果你在寻找某个特定的 Sun 规格说明对应的 Apache Geronimo 项目，我们已经在附录归纳了一个可用规格说明的列表。

这里还有必要指出的是我们的这个依赖使用了 provided 范围。这个范围告诉 Maven jar 文件已经由容器“提供”了，因此不再需要包含在 war 中。

如果你对在这个简单 web 应用编写自定义 JSP 标签感兴趣，你将需要添加对 JSP 2.0 规格说明的依赖。 使用以下配置来加入这个依赖。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

例 5.8. 添加 JSP 2.0 规格说明作为依赖

```
<project>
  [...]
  <dependencies>
    [...]
    <dependency>
      <groupId>org.apache.geronimo.specs</groupId>
      <artifactId>geronimo-jsp_2.0_spec</artifactId>
      <version>1.1</version>
      <scope>provided</scope>
    </dependency>
  </dependencies>
  [...]
</project>
```

在添加好这个 Servlet 规格说明依赖之后，运行 mvn clean install ，然后运行 mvn jetty:run 。

```
[tobrien@t1 simple-webapp]$ mvn clean install
...
[tobrien@t1 simple-webapp]$ mvn jetty:run
[INFO] [jetty:run]
...
2007-12-14 16:18:31.305::INFO:  jetty-6.1.6rc1
2007-12-14 16:18:31.453::INFO:  No Transaction manager found - if your webapp requi
           please configure one.
2007-12-14 16:18:32.745::INFO:  Started SelectChannelConnector@0.0.0.0:8080
[INFO] Started Jetty Server
```

到此为止，你应该能够获取 SimpleServlet 的输出。 在命令行，你可以使用 curl 在标准输出打印 servlet 的输出。

```
~/examples$ curl http://localhost:8080/simple-webapp/simple
SimpleServlet Executed
```

5.7. 小结

阅读完本章之后，你应该能够启动一个简单的 web 应用程序。 本章并没有详细描述用很多不同的方式来创建一个完整的 web 引用，其它章节对那些包含很多流行 web 框架和技术的项目提供了更全面的介绍。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

第 6 章 一个多模块项目

6.1. 简介

本章，我们创建一个结合了前两章样例的多模块项目。???中开发的simple-weather代码将会与第 5 章一个简单的Web应用中定义的simple-webapp结合以创建一个新的web应用，它获取天气预报信息然后显示在web页面上。本章最后，你能将够使用Maven开发复杂的，多模块项目。

6.1.1. 下载本章样例

该样例中开发的多模块项目包含了???和第 5 章一个简单的Web应用中项目的修改的版本，我们不会再使用Maven Archetype插件来生成这个多模块项目。我们强烈建议当你在阅读本章内容的时候，下载样例代码作为一个补充参考。本章的样例项目包含在本书的样例代码中，你可以从两个地方下载，http://www.sonatype.com/book/mvn-examples-1.0.zip或者http://www.sonatype.com/book/mvn-examples-1.0.tar.gz。解开存档文件至任意目录，然后到ch06/目录。在ch06/目录你会看到一个名为simple-parent/的目录，它包含了本章开发出来的多模块Maven项目。在这个simple-parent/项目目录中，你会看到一个pom.xml，以及两个子模块目录simple-weather/和simple-webapp/。如果你想要在浏览器里看样例代码，访问http://www.sonatype.com/book/examples-1.0，然后点击ch06/目录。

6.2. simple-parent 项目

一个多模块项目通过一个父POM引用一个或多个子模块来定义。在simple-parent/目录中你能找到一个父POM（也称为顶层POM）为simple-parent/pom.xml。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

例 6.1. simple-parent 项目的 POM

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                      http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>org.sonatype.mavenbook.ch06</groupId>
  <artifactId>simple-parent</artifactId>
  <packaging>pom</packaging>
  <version>1.0</version>
  <name>Chapter 6 Simple Parent Project</name>

  <modules>
    <module>simple-weather</module>
    <module>simple-webapp</module>
  </modules>

  <build>
    <pluginManagement>
      <plugins>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-compiler-plugin</artifactId>
          <configuration>
            <source>1.5</source>
            <target>1.5</target>
          </configuration>
        </plugin>
      </plugins>
   </pluginManagement>
  </build>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

注意simple-parent定义了一组Maven坐标：groupId是org.sonatype.mavenbook，artifactId是simple-parent，version是1.0。这个父项目不像之前的项目那样创建一个JAR或者一个WAR，它仅仅是一个引用其它Maven项目的POM。像simple-parent这样仅仅提供项目对象模型的项目，正确的的打包类型是pom。pom.xml中下一部分列出了项目的子模块。这些模块在modules元素中定义，每个modules元素对应了一个simple-parent/目录下的子目录。Maven知道去这些子目录寻找pom.xml文件，并且，在构建的simp-parent的时候，它会将这些子模块包含到要构建的项目中。

最后，我们定义了一些将会被所有子模块继承的设置。simple-parent的build部分配置了所有Java编译的目标是Java 5 JVM。因为compiler插件默认绑定到了生命周期，我们就可以使用pluginManagement部分来配置。我们将会在后面的章节详细讨论pluginManagement，区分为默认的插件提供配置和真正的绑定插件是很容易的。dependencies元素将JUnit 3.8.1添加为一个全局的依赖。build配置和dependencies都会被所有的子模块继承。使用POM继承允许你添加一些全局的依赖如JUnit和Log4J。

6.3. simple-weather 模块

我们要看的第一个子模块是simple-weather子模块。这个子模块包含了所有用来与Yahoo! weather信息源交互的类。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

```
    <version>1.0</version>
  </parent>
  <artifactId>simple-weather</artifactId>
  <packaging>jar</packaging>

  <name>Chapter 6 Simple Weather API</name>

  <build>
    <pluginManagement>
      <plugins>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-surefire-plugin</artifactId>
          <configuration>
            <testFailureIgnore>true</testFailureIgnore>
          </configuration>
        </plugin>
      </plugins>
    </pluginManagement>
  </build>

  <dependencies>
    <dependency>
      <groupId>log4j</groupId>
      <artifactId>log4j</artifactId>
      <version>1.2.14</version>
    </dependency>
    <dependency>
      <groupId>dom4j</groupId>
      <artifactId>dom4j</artifactId>
      <version>1.6.1</version>
    </dependency>
    <dependency>
      <groupId>jaxen</groupId>
      <artifactId>jaxen</artifactId>
      <version>1.1.1</version>
    </dependency>
    <dependency>
      <groupId>velocity</groupId>
      <artifactId>velocity</artifactId>
      <version>1.5</version>
    </dependency>
    <dependency>
      <groupId>org.apache.commons</groupId>
      <artifactId>commons-io</artifactId>
      <version>1.3.2</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

在simple-weather的pom.xml文件中我们看到该模块使用一组Maven坐标引用了一个父POM。simple-weather的父POM通过一个值为org.sonatype.mavenbook的groupId，一个值为simple-parent的artifactId，以及一个值为1.0的version来定义。注意子模块中我们不再需要重新定义groupId和version，它们都从父项目继承了。

例 6.3. WeatherService 类

```
package org.sonatype.mavenbook.weather;

import java.io.InputStream;

public class WeatherService {

    public WeatherService() {}

    public String retrieveForecast( String zip ) throws Exception {
        // Retrieve Data
        InputStream dataIn = new YahooRetriever().retrieve( zip );

        // Parse Data
        Weather weather = new YahooParser().parse( dataIn );

        // Format (Print) Data
        return new WeatherFormatter().format( weather );
    }
}
```

WeatherService类在src/main/java/org/sonatype/mavenbook/weather中定义，它简单的调用???中定义的三个对象。在本章的样例中，我们正创建一个单独的项目，它包含了将会在web应用项目中被引用的service对象。这是一个在企业级Java开发中常见的模型，通常一个复杂的应用包含了不止一个的简单web应用。你可能拥有一个企业应用，它包含了多个web应用，以及一些命令行应用。通常你会想要重构那些通用的逻辑至一个service类，以被很多项目重用。这就是我们创建WeatherService类的理由，在此之后，你就能看到simple-webapp项目是如何引用在simple-weather中定义的service对象。

retrieveForecast()方法接受一个包含邮政编码的String。之后这个邮政编码参数被传给YahooRetriever的retrieve()方法，后者从Yahoo! Weather获取XML。从YahooRetriever返回的XML被传给YahooParser的parse()方法，后者继而又返回一个Weather对象。之后这个Weather对象被WeatherFormatter格式化成一个像样的String。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

6.4. simple-webapp 模块

simple-webapp模块是在simple-parent项目中引用的第二个子模块。这个web项目依赖于simple-weather模块，并且包含了一些用来展示从Yahoo! Weather服务查询到的结果的servlet。

例 6.4. simple-webapp 模块的 POM

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                      http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.sonatype.mavenbook.ch06</groupId>
    <artifactId>simple-parent</artifactId>
    <version>1.0</version>
  </parent>

  <artifactId>simple-webapp</artifactId>
  <packaging>war</packaging>
  <name>simple-webapp Maven Webapp</name>
  <dependencies>
    <dependency>
      <groupId>org.apache.geronimo.specs</groupId>
      <artifactId>geronimo-servlet_2.4_spec</artifactId>
      <version>1.1.1</version>
    </dependency>
    <dependency>
      <groupId>org.sonatype.mavenbook.ch06</groupId>
      <artifactId>simple-weather</artifactId>
      <version>1.0</version>
    </dependency>
  </dependencies>
  <build>
    <finalName>simple-webapp</finalName>
    <plugins>
      <plugin>
        <groupId>org.mortbay.jetty</groupId>
        <artifactId>maven-jetty-plugin</artifactId>
      </plugin>
    </plugins>
  </build>
</project>
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

simple-weather模块定义了一个十分简单的servlet，它从HTTP请求读取一个邮政编码，调用例 6.3 “WeatherService 类”中展示的WeatherService，然后将结果打印至HTTP响应的Writer。

例 6.5. simple-webapp 的 WeatherServlet

```
package org.sonatype.mavenbook.web;

import org.sonatype.mavenbook.weather.WeatherService;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class WeatherServlet extends HttpServlet {
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {
        String zip = request.getParameter("zip" );
        WeatherService weatherService = new WeatherService();
        PrintWriter out = response.getWriter();
        try {
            out.println( weatherService.retrieveForecast( zip ) );
        } catch( Exception e ) {
            out.println( "Error Retrieving Forecast: " + e.getMessage() );
        }
        out.flush();
        out.close();
    }
}
```

在WeatherServlet中，我们初始化了一个在simple-weather中定义的WeatherService类的实例。在请求参数中提供的邮政编码被传给retrieveForecast()方法，并且返回结果被打印至HTTP响应的Writer。

最后，src/main/webapp/WEB-INF目录下的web.xml将所有这一切绑在一起。web.xml中的servlet和servlet-mapping元素将路径/weather匹配至WeatherServlet。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

例 6.6. simple-webapp 的 web.xml

```
<!DOCTYPE web-app PUBLIC
 "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
 "http://java.sun.com/dtd/web-app_2_3.dtd" >

<web-app>
  <display-name>Archetype Created Web Application</display-name>
  <servlet>
    <servlet-name>simple</servlet-name>
    <servlet-class>org.sonatype.mavenbook.web.SimpleServlet</servlet-class>
  </servlet>
  <servlet>
    <servlet-name>weather</servlet-name>
    <servlet-class>org.sonatype.mavenbook.web.WeatherServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>simple</servlet-name>
    <url-pattern>/simple</url-pattern>
  </servlet-mapping>
  <servlet-mapping>
    <servlet-name>weather</servlet-name>
    <url-pattern>/weather</url-pattern>
  </servlet-mapping>
</web-app>
```

6.5. 构建这个多模块项目

既然simple-weather项目包含了所有与Yahoo! Weather服务交互的代码，以及simple-webapp项目包含了一个简单的servlet，是时间将这个应用编译并打包成一个WAR文件了。为此，你会想要以合适的顺序编译并安装这两个项目；以为simple-webapp依赖于simple-weather，simple-weather的JAR需要在simple-webapp项目被编译之前就被创建好。为此，你需要从simple-parent项目运行mvn clean install命令。

```
~/examples/ch06/simple-parent$ mvn clean install
[INFO] Scanning for projects...
[INFO] Reactor build order:
[INFO]   Simple Parent Project
[INFO]   simple-weather
[INFO]   simple-webapp Maven Webapp
[INFO] ----------------------------------------------------------------------
[INFO] Building simple-weather
[INFO]    task-segment: [clean, install]
[INFO] ----------------------------------------------------------------------
[...]
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛


```
[INFO] [install:install]
[INFO] Installing simple-weather-1.0.jar to simple-weather-1.0.jar
[INFO] ------------------------------------------------------------------------
[INFO] Building simple-webapp Maven Webapp
[INFO]    task-segment: [clean, install]
[INFO] ------------------------------------------------------------------------
[...]
[INFO] [install:install]
[INFO] Installing simple-webapp.war to simple-webapp-1.0.war
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Reactor Summary:
[INFO] ------------------------------------------------------------------------
[INFO] Simple Parent Project ............................... SUCCESS [3.041s]
[INFO] simple-weather ...................................... SUCCESS [4.802s]
[INFO] simple-webapp Maven Webapp .......................... SUCCESS [3.065s]
[INFO] ------------------------------------------------------------------------
```

当Maven执行一个带有子模块的项目的时候，Maven首先载入父POM，然后定位所有的子模块POM。Maven然后将所有这些项目的POM放入到一个称为Maven 反应堆（Reactor）的东西中，由它负责分析模块之间的依赖关系。这个反应堆处理组件的排序，以确保相互独立的模块能以适当的顺序被编译和安装。

注意

除非需要做更改，反应堆一直维持定义在POM中的模块的顺序。为此一个有帮助的思维模型是，那些依赖于兄弟项目的项目在列表中被“向下按”，直到依赖顺序被满足。在很少的情况下，重新安排你构建的模块顺序可能很方便——例如你想要一个频繁的不稳定的模块接近构建的开端。

一旦反应堆解决了项目构建的顺序，Maven就会在多模块构建中为每个模块执行特定的目标。本例中，你能看到Maven在simple-webapp之前构建了simple-weather，为每个子模块执行了mvn clean install。

注意

当你在命令行运行Maven的时候你经常会在任何其它生命周期阶段前指定clean生命周期阶段。当你指定clean，你就确认了在编译和打包一个应用之前，Maven会移除旧的输出。运行clean不是必要的，但这是一个确保你正执行“干净构建”的十分有用的预防措施。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

6.6. 运行Web应用

一旦这个多模块项目已经通过从父项目simple-project执行mvn clean install行安装好了，你可以切换目录至simple-webapp项目，然后运行Jetty插件的Run目标。

```
~/examples/ch06/simple-parent/simple-webapp $ mvn jetty:run
[INFO] ----------------------------------------------------------------------
[INFO] Building simple-webapp Maven Webapp
[INFO]    task-segment: [jetty:run]
[INFO] ----------------------------------------------------------------------
[...]
[INFO] [jetty:run]
[INFO] Configuring Jetty for project: simple-webapp Maven Webapp
[...]
[INFO] Webapp directory = ~/examples/ch06/simple-parent/\
                          simple-webapp/src/main/webapp
[INFO] Starting jetty 6.1.6rc1 ...
2007-11-18 1:58:26.980::INFO:  jetty-6.1.6rc1
2007-11-18 1:58:26.125::INFO:  No Transaction manager found
2007-11-18 1:58:27.633::INFO:  Started SelectChannelConnector@0.0.0.0:8080
[INFO] Started Jetty Server
```

Jetty启动之后，在浏览器载入http://localhost:8080/simple-webapp/weather?zip=01201，你应该看到格式化的天气输出。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

第 7 章 多模块企业级项目

7.1. 简介

本章，我们创建一个多模块项目，它从第 6 章一个多模块项目和第 5 章一个简单的Web应用的样例演化成一个使用了Spring Framework和Hibernate创建的，从Yahoo! Weather信息源读取数据，包含一个简单web应用和一个命令行工具的项目。???中开发的simple-weather代码将会和第 5 章一个简单的Web应用中开发的simple-weather项目结合。在创建这个多模块项目的过程中，我们将会探索Maven并且讨论用不同方式来创建模块化项目以鼓励重用。

7.1.1. 下载本章样例

该样例中开发的多模块项目包含了???和第 5 章一个简单的Web应用中项目的修改的版本，我们不会再使用Maven Archetype插件来生成这个多模块项目。我们强烈建议当你在阅读本章内容的时候，下载样例代码作为一个补充参考。本章的样例项目包含在本书的样例代码中，你可以从两个地方下载，http://www.sonatype.com/book/mvn-examples-1.0.zip或者http://www.sonatype.com/book/mvn-examples-1.0.tar.gz。解开存档文件至任意目录，然后到ch07/目录。在ch07/目录你会看到一个名为simple-parent/的目录，它包含了本章开发出来的多模块Maven项目。在这个simple-parent/项目目录中，你会看到一个pom.xml，以及五个子模块目录simple-model/，simple-persist/，simple-command/，simple-weather/和simple-webapp/。如果你想要在浏览器里看样例代码，访问http://www.sonatype.com/book/examples-1.0，然后点击ch07/目录。

7.1.2. 多模块企业级项目

展示一个巨大企业级项目的复杂度远远超出了本书的范围。这样的项目的特征有，多数据库，与外部系统的集成，子项目通过部门来划分。这些项目通常跨越了数千行代码，牵涉了数十或数百软件开发者的努力。虽然这样的完整样例超出了本书的范围，我们仍然可以为你提供一个能让你想起大型企业应用的样例项目。在小结中我们提议了一些在本章描述之外的模块化可能性。

本章，我们将会看一个多模块Maven项目，它将产生两个应用程序：一个对于Yahoo! Weather信息源的命令行查询工具，以及查询同样信息源的一个web应用。两个应用都会将查询结果存储到一个内嵌数据库中。都允许用户从内嵌数据库中获取历史天气数据。都会重用应用程序逻辑，并且共享一个持久化类库。本章样例基于在???中介绍的Yahoo! Weather解析代码构建。该项目被划分成如图 7.1 “多模块企业级应用的模块关系”所示的五个子项目。

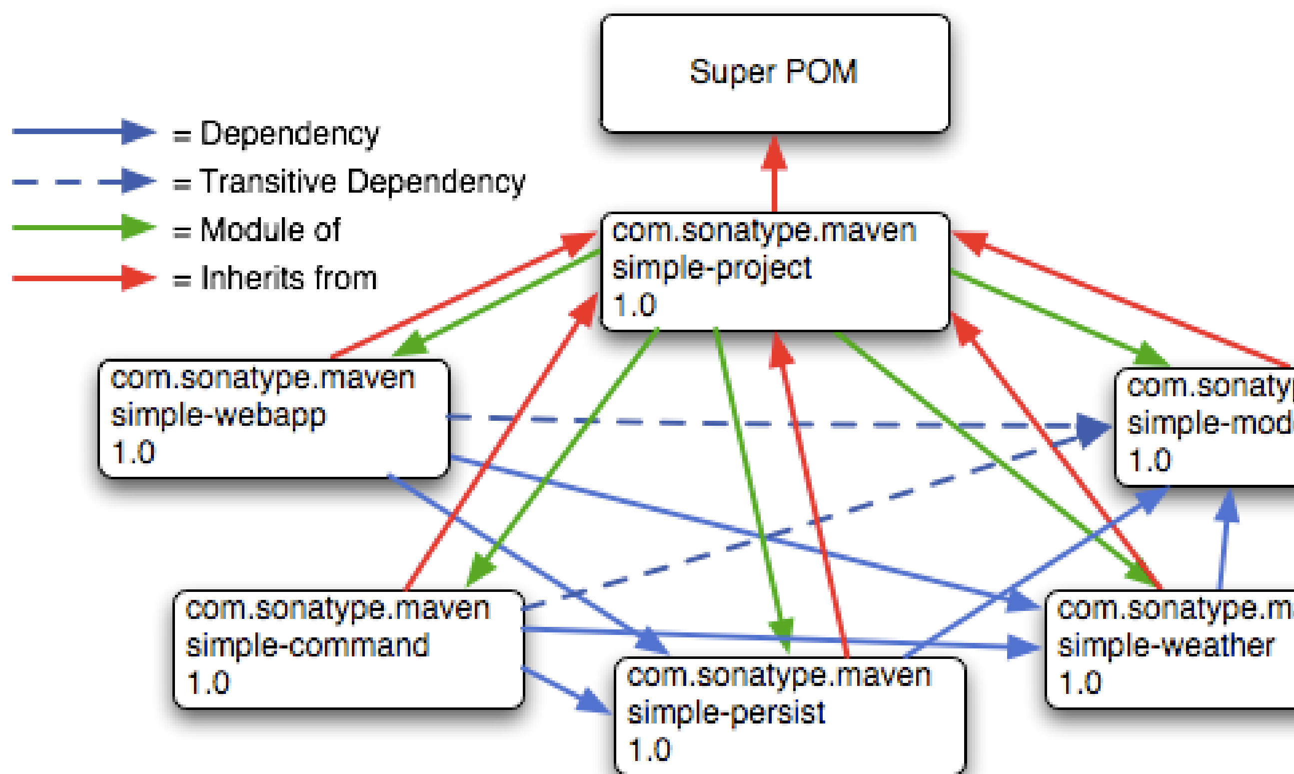


图 7.1. 多模块企业级应用的模块关系

在图 7.1 “多模块企业级应用的模块关系”中，你能看到simple-parent有五个子模块，它们分别是：

simple-model
: 该模块定义了一个简单的对象模型，对从Yashoo! Weather信息源返回的数据建模。该对象模型包含了Weather，Condition，Atmosphere，Location，和Wind对象。当我们的应用程序解析Yahoo! Weather信息源的时候，simple-weather中定义的解析器会解析XML并创建供应用程序使用的Weather对象。该项目还包含了使用Hibernate 3标注符标注的模型对象，它们在simple-persist的逻辑中被用来映射每个模型对象至关系数据库中对应的表。

simple-weather
: 该模块包含了所有用来从Yahoo! Weather数据源获取数据并解析结果XML的逻辑。从数据源返回的XML被转换成simple-model中定义的模型对象。simple-weather有一个对simple-model的依赖。simple-weather定义了一个WeatherService对象，该对象会被simple-command和simple-webapp项目引用。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

simple-persist
: 该模块包含了一些数据访问对象(DAO)，这些对象将Weather对象存储在一个内嵌数据库中。这个多模块项目中的两个应用都会使用simple-persist中定义的DAO来将数据存储至内嵌数据库中。本项目中定义的DAO能理解并返回simple-model定义的模型对象。simple-persist有一个对simple-model的依赖，它也依赖于模型对象上的Hibernate标注。

simple-webapp
: 这个web应用项目包含了两个Spring MVC控制器实现，控制器使用了simple-weather中定义的WeatherService，以及simple-persist中定义的DAO。simple-webapp有对于simple-weather和simple-persist的直接依赖；还有一个对于simple-model的传递性依赖。

simple-command
: 该模块包含了一个用来查询Yahoo! Weather信息源的简单命令行工具。它包含了一个带有静态main()方法的类，与simple-weather中定义的WeatherService和simple-persist中定义的DAO交互。simple-command有对于simple-weather和simple-persist的直接依赖；还有一个对于simple-model的传递性依赖。

本章设计的项目一方面够简单，以能在一本书中介绍，又够复杂，能提供一组五个子模块。该样例有一个带有五个类的模型项目，带有两个服务类的持久化类库，带有五六个类的天气解析类库，但是一个现实系统可能有一个带有数百对象的模型项目，很多持久化类库，以及跨越多个部门的服务类库。虽然我们试图确保本例中的代码尽可能的直接以能在短时间内理解，但我们也不怕麻烦的以模块化的方式构建了这个项目。你可能会要看一下本章的样例，然后会认为Maven为我们这个只有五个类的模型项目带来了太多的复杂度。虽然使用Maven确实建议一定程度的模块化，但这里我们不怕麻烦的将样例项目弄得复杂，目的是展示Maven的多模块特性。

7.1.3. 本例中所用的技术

本章样例中涉及了一些十分流行，但与Maven没有直接关系的而技术。这些技术是Spring Framework和Hibernate。Spring Framework是一个反转控制(IoC)容器，以及一组目的在于简化与各种J2EE类库交互的框架。使用Spring Framework作为应用程序开发的基础框架能让你访问很多有用的抽象接口，它们能简化与持久化框架如Hibernate或者iBatis的交互，以及企业API如JDBC，JNDI，和JMS。Spring Framework在过去一些年变得十分流行，作为对来自Sun微系统的重量级企业标准的替代。Hibernate是一个被广泛使用的对象-关系映射框架，能让你与关系数据库的交互就像它们是Java对象的集合一样。本例关注构建一个简单的web应用和一个命令行应用，它们使用Spring Framework为应用暴露了一组可重用的组件，使用Hibernate将天气数据持久化至内嵌数据库。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

我们决定包含对这些框架的参考以展示在使用Maven的时候如何使用这些技术构建项目。虽然本章中我们会大概介绍这些技术，但不是完整的解释这些技术。要了解更多关于Spring Framework的信息，请查看该项目的web站点：http://www.springframework.org/。要了解更多关于Hibernate和Hibernate标注的信息，请查看该项目的web站点：http://www.hibernate.org。本章使用了HSQLDB作为一个内嵌数据库；要了解更多的关于该数据库的信息，访问该项目的web站点：http://hsqldb.org/。

7.2. simple-parent项目

#simple-parent##### pom.xml ，它引用了五个子模块：simple-command，simple-model，simple-weather，simple-persist，和simple-webapp。顶层的pom.xml在例 7.1 “simple-parent 项目的 POM”中显示。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛


```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                      http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>org.sonatype.mavenbook.ch07</groupId>
  <artifactId>simple-parent</artifactId>
  <packaging>pom</packaging>
  <version>1.0</version>
  <name>Chapter 7 Simple Parent Project</name>

  <modules>
    <module>simple-command</module>
    <module>simple-model</module>
    <module>simple-weather</module>
    <module>simple-persist</module>
    <module>simple-webapp</module>
  </modules>

  <build>
    <pluginManagement>
      <plugins>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-compiler-plugin</artifactId>
          <configuration>
            <source>1.5</source>
            <target>1.5</target>
          </configuration>
        </plugin>
      </plugins>
   </pluginManagement>
  </build>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

注意这个父POM和例 6.1 “simple-parent 项目的 POM”中定义的父POM的相似度。这两个POM唯一真正不同的地方是子模块列表。前面的样例中只列出了两个子模块，而这里的父POM列出了五个子模块。下面的小节会详细讨论所有这五个子模块。因为我们的样例使用了Java标注，我们将编译器的目标配置成Java 5 JVM。

7.3. simple-model模块

大多数企业项目需要做的第一件事情是建立对象模型。一个对象模型抓取了系统中一组核心的领域对象。一个银行系统可能会有包括Account，Customer，Transaction 的对象模型。或者有一个对体育比赛比分建模的系统，有一个Team和一个Game对象。不管它是什么，你都需要将你系统中的概念建模成对象模型。在Maven项目中，将这个对象模型分割成单独的项目以被广泛引用，是一种常用的实践。在我们这个系统中，我们将每个对Yahoo! Weather数据源的查询建模成为Weather对象，它本身又引用了四个其它的对象。风向，风速等存储Wind在对象中。地址信息包括邮编，城市等信息存储在Location类中。大气信息如湿度，可见度，气压等存储在Atmosphere类中。而对环境，气温，以及观察日期的文本描述存储在Condition类中。

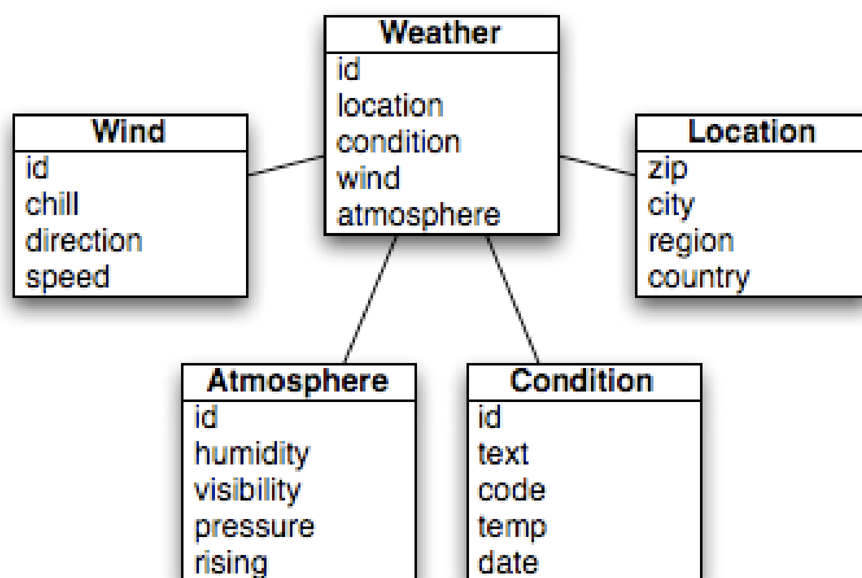


图 7.2. 天气数据的简单对象模型

这个简单对象模型的pom.xml文件含有一个依赖需要一些解释。我们的对象模型用Hibernate标注符标注了。我们用这些标注来映射模型对象至关系数据库中的表。这个依赖是org.hibernate:hibernate-annotations:3.3.0.ga。看一下例 7.2 “simple-model 的 pom.xml”中显示的pom.xml，然后看接下来几个展示这些标注的例子。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

例 7.2. simple-model 的 pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                      http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.sonatype.mavenbook.ch07</groupId>
    <artifactId>simple-parent</artifactId>
    <version>1.0</version>
  </parent>
  <artifactId>simple-model</artifactId>
  <packaging>jar</packaging>

  <name>Simple Object Model</name>

  <dependencies>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-annotations</artifactId>
      <version>3.3.0.ga</version>
    </dependency>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-commons-annotations</artifactId>
      <version>3.3.0.ga</version>
    </dependency>
  </dependencies>
</project>
```

在src/main/java/org/sonatype/mavenbook/weather/model中，有Weather.java，它是一个标注过的Weather型对象。这个Weather对象是一个简单的Java bean。这意味着它的私有成员变量如id, location, condition, wind, atmosphere, 和 date，通过公共的getter和setter方法暴露，并且遵循这样的模式：如果一个属性名为name，那么会有一个公有的无参数方法getName()，还有一个带有一个参数的setter方法setName(String name)。我们只是展示了id属性的getter和setter方法，其它属性的getter和setter方法类似，所以这里跳过了，以节省篇幅。请看例 7.3 “标注的Weather模型对象”。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

例 7.3. 标注的Weather模型对象

```
package org.sonatype.mavenbook.weather.model;

import javax.persistence.*;

import java.util.Date;

@Entity
@NamedQueries({
  @NamedQuery(name="Weather.byLocation",
              query="from Weather w where w.location = :location")
})
public class Weather {

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Integer id;

    @ManyToOne(cascade=CascadeType.ALL)
    private Location location;

    @OneToOne(mappedBy="weather",cascade=CascadeType.ALL)
    private Condition condition;

    @OneToOne(mappedBy="weather",cascade=CascadeType.ALL)
    private Wind wind;

    @OneToOne(mappedBy="weather",cascade=CascadeType.ALL)
    private Atmosphere atmosphere;

    private Date date;

    public Weather() {}

    public Integer getId() { return id; }
    public void setId(Integer id) { this.id = id; }

    // All getter and setter methods omitted...
}
```

在Weather类中，我们使用Hibernate标注以为simple-persist项目提供指导。这些标注由Hibernate用来将对象与关系数据库映射。尽管对HIberante标注的完整解释超出了本书的范围，这里是一个为好奇者的简单解释。@Entity标注标记一个类为持久化对象。我们省略了这个类的@Table标注，因此Hibernate将会使用这个类的名字

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

来映射表名。@NamedQueries注解定义了一个simple-persist中WeatherDAO使用的查询。@NamedQuery注解中的查询语句是用一个叫做Hibernate查询语言(HQL)编写的。每个成员变量的注解定义了这一列的类型，以及该列暗示的表关联关系。

Id
: id属性用@Id进行标注。这标记id属性为一个包含数据库表主键的属性。@GeneratedValue控制新的主键值如何产生。该例中，我们使用IDENTITY GenerationType，它使用了下层数据库的主键生成设施。

Location
: 每个Weather对象实例对应了一个Location对象。一个Location对象含有一个邮政编码，而@ManyToOne确认所有指向同一个Location对象的Weather对象引用了同样一个实例。@ManyToOne的cascade属性确保每次我们持久化一个Weather对象的时候也会持久化一个Location对象。

Condition, Wind, Atmosphere
: 这些对象的每一个都作为@OneToOne而且CascadeType为ALL进行映射。这意味着每次我们保存一个Weather对象，我们将会往Weather表，Condition表，Wind表，和Atmosphere表，插入一行，

Date
: Date没有被标注，这以为着Hibernate将会使用所有列的默认值来定义该映射。列名将会是date，列的类型会是匹配Date对象的适当时间。

注意

如果你有一个希望从表映射中忽略的属性，你可以使用@Transient标注这个属性。

接着，看一下一个二级的模型对象，Condition，如例 7.4 “simple-model 的 Condition 模型对象”所示。这个类同样也存在于src/main/java/org/sonatype/mavenbook/weather/model。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

例 7.4. simple-model 的 Condition 模型对象

```
package org.sonatype.mavenbook.weather.model;

import javax.persistence.*;

@Entity
public class Condition {

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Integer id;

    private String text;
    private String code;
    private String temp;
    private String date;

    @OneToOne(cascade=CascadeType.ALL)
    @JoinColumn(name="weather_id", nullable=false)
    private Weather weather;

    public Condition() {}

    public Integer getId() { return id; }
    public void setId(Integer id) { this.id = id; }

    // All getter and setter methods omitted...
}
```

这个`Condition`类类似于`Weather`类。它被标注为一个`@Entity`，在`id`属性上也有相似的标注。`text`，`code`，`temp`，和`date`属性也都使用默认的列设置，`weather`属性使用了`@OneToOne`进行标注，而另一个标注通过一个名为`weather_id`的外键引用关联的`Weather`对象。

7.4. simple-weather模块

我们将要检查的下一个模块可以被认为是一个“服务”。这个simple-weather模块包含了所有从Yahoo! Weather RSS数据源获取数据并解析的必要逻辑。虽然simple-weather只包含了三个Java类和一个JUnit测试，它还将展现为一个单独的组件，`WeatherService`，同时为简单web应用和简单命令行工具服务。通常来说一个企业级项目会包含一些API模块，这写模块包含了重要的业务逻辑，或者与外部系统交互的逻辑。一个银行系统可能有一个模块，从第三方数据提供者获取并解析数据，而一个显示赛事比分的系统可能会与一个提供实时篮球或足球比分的XML数据源进行交互。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

在例 7.5 “simple-weather 模块的 POM”中，该模块封装了所有的网络活动，以及与Yahoo! Weather交互涉及的XML解析活动。其它依赖于该模块的项目只要简单的调用WeatherService的retrieveForecast()方法，该方法接受一个邮政编码作为参数，返回一个Weather对象。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛


```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                      http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.sonatype.mavenbook.ch07</groupId>
    <artifactId>simple-parent</artifactId>
    <version>1.0</version>
  </parent>
  <artifactId>simple-weather</artifactId>
  <packaging>jar</packaging>

  <name>Simple Weather API</name>

  <dependencies>
    <dependency>
      <groupId>org.sonatype.mavenbook.ch07</groupId>
      <artifactId>simple-model</artifactId>
      <version>1.0</version>
    </dependency>
    <dependency>
      <groupId>log4j</groupId>
      <artifactId>log4j</artifactId>
      <version>1.2.14</version>
    </dependency>
    <dependency>
      <groupId>dom4j</groupId>
      <artifactId>dom4j</artifactId>
      <version>1.6.1</version>
    </dependency>
    <dependency>
      <groupId>jaxen</groupId>
      <artifactId>jaxen</artifactId>
      <version>1.1.1</version>
    </dependency>
    <dependency>
      <groupId>org.apache.commons</groupId>
      <artifactId>commons-io</artifactId>
      <version>1.3.2</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

这个simple-weather POM扩展了simple-parent POM，设置打包方式为jar，然后添加了下列的依赖：

org.sonatype.mavenbook.ch07:simple-model:1.0
: simple-weather将Yahoo! Weather RSS数据源解析成一个Weather对象。 它有一个对simple-model的直接依赖。.

log4j:log4j:1.2.14
: simple-weather使用Log4J类库来打印日志信息。

dom4j:dom4j:1.6.1 and jaxen:jaxen:1.1.1
: 这两个依赖都用来解析从Yahoo! Weather返回的XML。

org.apache.commons:commons-io:1.3.2 (scope=test)
: 这个test范围的依赖是由YahooParserTest使用的。

接下来是WeatherService类，在例 7.6 “WeatherService 类”中显示。这个类和例 6.3 “WeatherService 类”中的WeatherService类看起来很像。虽然WeatherService名字一样，但与本章中的样例还是有细微的差别。这个版本的retrieveForecast()方法返回一个Weather对象，而格式就留给调用WeatherService的程序去处理。其它的主要变化是，YahooRetriever和YahooParser都是WeatherService bean的bean属性。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

例 7.6. WeatherService 类

```
package org.sonatype.mavenbook.weather;

import java.io.InputStream;

import org.sonatype.mavenbook.weather.model.Weather;

public class WeatherService {

  private YahooRetriever yahooRetriever;
  private YahooParser yahooParser;

  public WeatherService() {}

  public Weather retrieveForecast(String zip) throws Exception {
    // Retrieve Data
    InputStream dataIn = yahooRetriever.retrieve(zip);

    // Parse DataS
    Weather weather = yahooParser.parse(zip, dataIn);

    return weather;
  }

  public YahooRetriever getYahooRetriever() {
    return yahooRetriever;
  }

  public void setYahooRetriever(YahooRetriever yahooRetriever) {
    this.yahooRetriever = yahooRetriever;
  }

  public YahooParser getYahooParser() {
    return yahooParser;
  }

  public void setYahooParser(YahooParser yahooParser) {
    this.yahooParser = yahooParser;
  }
}
```

最后，在这个项目中我们有一个由Spring Framework用来创建ApplicationContext的XML文件。首先，一些解释：两个应用程序，web应用和命令行工具，都需要和WeatherService类交互，而且它们都使用名字weatherService从

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

Spring ApplicationContext获取此类的一个实例。我们的web应用使用一个与WeatherService实例关联的Spring MVC控制器，我们的命令行应用在静态main()方法中从ApplicationContext载入这个WeatherService。为了鼓励重用，我们已经在src/main/resources中包含了一个applicationContext-weather.xml文件，这样便在classpath中可用。依赖于simple-weather模块的模块可以使用Spring Framework中的ClasspathXmlApplicationContext载入这个Application Context。之后它们就能引用名为weatherService的WeatherService实例。

例 7.7. simple-weather模块的Spring Application Context

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
             http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
    default-lazy-init="true">

    <bean id="weatherService"
             class="org.sonatype.mavenbook.weather.WeatherService">
      <property name="yahooRetriever" ref="yahooRetriever"/>
      <property name="yahooParser" ref="yahooParser"/>
    </bean>

    <bean id="yahooRetriever"
             class="org.sonatype.mavenbook.weather.YahooRetriever"/>

    <bean id="yahooParser"
             class="org.sonatype.mavenbook.weather.YahooParser"/>
</beans>
```

该文档定义了三个bean：yahooParser，yahooRetriever，和weatherService。weatherService bean是WeatherService的一个实例，这个XML文档通过引用对应类的命名实例来填充yahooParser和yahooRetriever属性。可以将这个applicationContext-weather.xml文件看作是定义了这个多模块项目中一个子系统的架构。其它项目如simple-webapp和simple-command可以引用这个上下文，获取一个已经建立好与YahooRetriever和YahooParser实例关系的WeatherService实例。

7.5. simple-persist模块

该模块定义了两个简单的数据访问对象（DAO）。一个DAO是一个提供持久化操作接口的对象。在这个应用中我们使用了对象关系映射（ORM）框架Hibernate，DAO通常在对象旁边定义。在本项目中，我们定义两个DAO对象：WeatherDAO和LocationDAO。WeatherDAO类允许我们保存一个Weather对象

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

至数据库，根据id获取一个Weather对象，获取匹配特定Location的Weather对象。LocationDAO有一个方法允许我们根据邮政编码获取Location对象。首先，让我们看一下例 7.8 “simple-persist 的 POM”中的simple-persist POM。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

```
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.sonatype.mavenbook.ch07</groupId>
    <artifactId>simple-parent</artifactId>
    <version>1.0</version>
  </parent>
  <artifactId>simple-persist</artifactId>
  <packaging>jar</packaging>

  <name>Simple Persistence API</name>

  <dependencies>
    <dependency>
      <groupId>org.sonatype.mavenbook.ch07</groupId>
      <artifactId>simple-model</artifactId>
      <version>1.0</version>
    </dependency>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate</artifactId>
      <version>3.2.5.ga</version>
      <exclusions>
        <exclusion>
          <groupId>javax.transaction</groupId>
          <artifactId>jta</artifactId>
        </exclusion>
      </exclusions>
    </dependency>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-annotations</artifactId>
      <version>3.3.0.ga</version>
    </dependency>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-commons-annotations</artifactId>
      <version>3.3.0.ga</version>
    </dependency>
    <dependency>
      <groupId>org.apache.geronimo.specs</groupId>
      <artifactId>geronimo-jta_1.1_spec</artifactId>
      <version>1.1</version>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring</artifactId>
      <version>2.0.7</version>
    </dependency>
  </dependencies>
</project>
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

这个POM文件引用simple-parent作为一个父POM，它定义了一些依赖。simple-persist的POM中列出的依赖有：

org.sonatype.mavenbook.ch07:simple-model:1.0
: 就像simple-weather模块一样，这个持久化模块引用了simple-model中定义的核心模型对象。

org.hibernate:hibernate:3.2.5.ga
: 我们定义了一个对HIbernate版本3.2.5ga的依赖，但注意我们排除了Hibernate的一个依赖。这么做是因为javax.transaction:javax依赖在公共Maven仓库中不可用。此依赖正好是Sun依赖中的一个，不能免费在中央Maven仓库中提供。为了避免烦人的信息告诉我们去下载非免费的依赖，我们简单的从Hibernate排除这个依赖然后添加一个geronimo-jta_1.1_spec依赖。

org.apache.geronimo.specs:geronimo-jta_1.1_spec:1.1
: 就像Servlet和JSP API，Apache Geronimo项目也根据Apache许可证友好的发布了一些认证过的企业级API。这意味着不管什么时候某个组件告诉你它依赖于JDBC，JNDI，和JTA API，你都可以查一下groupId为org.apache.geronimo.specs下的对应类库。

org.springframework:spring:2.0.7
: 这里包含了整个Spring Framework作为一个依赖。

注意

只依赖于你正使用的Spring组件是一个很好的实践。Spring Framework项目很友好的创建了一些有针对性的构件如spring-hibernate3。

为什么依赖于Spring呢？当和Hibernate集成的时候，Spring允许我们使用一些帮助类如HibernateDaoSupport。作为一个HibernateDaoSupport的样例，看一下例 7.9 “simple-persist'的WeatherDAO类”中的WeatherDAO代码。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

例 7.9. simple-persist'的WeatherDAO类

```
package org.sonatype.mavenbook.weather.persist;

import java.util.ArrayList;
import java.util.List;

import org.hibernate.Query;
import org.hibernate.Session;
import org.springframework.orm.hibernate3.HibernateCallback;
import org.springframework.orm.hibernate3.support.HibernateDaoSupport;

import org.sonatype.mavenbook.weather.model.Location;
import org.sonatype.mavenbook.weather.model.Weather;

public class WeatherDAO extends HibernateDaoSupport# {

    public WeatherDAO() {}

    public void save(Weather weather) {#
      getHibernateTemplate().save( weather );
    }

    public Weather load(Integer id) {#
      return (Weather) getHibernateTemplate().load( Weather.class, id);
    }

    @SuppressWarnings("unchecked")
    public List<Weather> recentForLocation( final Location location ) {
      return (List<Weather>) getHibernateTemplate().execute(
        new HibernateCallback() {#
        public Object doInHibernate(Session session) {
          Query query = getSession().getNamedQuery("Weather.byLocation");
          query.setParameter("location", location);
          return new ArrayList<Weather>( query.list() );
        }
      });
    }
}
```

就是这样。你已经编写了一个类，它能插入新的行，根据主键选取，以及能查找所有Weather表根据id连接Location表的结果。很显然，我们不能将书停在这里，然后花500页给你解释Hibernate的运作详情，但我们能做一些快速简单的解释：

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

❶ 继承HibernateDaoSupport的类。这个类会和Hibernate SessionFactory关联，后者将被用来创建Hibernate Session对象。在Hibernate中，每个操作都涉及Session对象，一个Session是访问下层数据库的中介，它也负责管理对DataSource的JDBC连接。继承HibernateDaoSupport也意味着我们能够使用getHibernateTemplate()访问HibernateTemplate。能使用HibernateTemplate完成的操作例子有……

❷ save()方法接受一个Weather实例然后调用HibernateTemplate上的save()方法。HibernateTemplate简化了常见的HIbernate操作的调用，并别将所有数据库特有的异常转换成了运行时异常。这里我们调用save()，它往Weather表中插入一条新的记录。可选的操作有update()，它更新已存在的一行，或者saveOrUpdate()，它会根据Weather中的non-null id属性是否存在，执行保存或者更新。

❸ load()方法，同样，也只是调用HibernateTemplate实例的方法。HibernateTemplate上的load()接受一个Class对象和一个Serializable对象。本例中，Serializable对应于要载入的Weather对象的id的值。

❹ 最后一个方法recentForLocation()调用定义在Weather模型对象中的NamedQuery。如果你的记忆力足够好，你就知道Weather模型对象定义了一个命名查询"Weather.byLocation"，查询为"from Weather w where w.location = :location"。我们通过HibernateCallback中的Hibernate Session对象来载入NamedQuery，HibernateCallback由HibernateTemplate的execute()方法执行。在这个方法中你能看到我们填充了一个命名参数location，它来自于recentForLocation()方法的参数。

现在是时候阐明一些情况了。HibernateDaoSupport和HibernateTemplate是来自于Spring Framework的类。它们由Spring Framework创建，目的是减少编写Hibernate DAO对象的痛苦。为了支持这个DAO，我们需要在simple-persist的Spring ApplicationContext定义中做一些配置。例 7.10 “simple-persist 的Spring Application Context”中显示的XML文档存储在src/main/resources，名为applicationContext-persist.xml。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

```
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.0.xsd"
    default-lazy-init="true">

    <bean id="sessionFactory"
class="org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean">
        <property name="annotatedClasses">
            <list>
                <value>org.sonatype.mavenbook.weather.model.Atmosphere</value>
                <value>org.sonatype.mavenbook.weather.model.Condition</value>
                <value>org.sonatype.mavenbook.weather.model.Location</value>
                <value>org.sonatype.mavenbook.weather.model.Weather</value>
                <value>org.sonatype.mavenbook.weather.model.Wind</value>
            </list>
        </property>
        <property name="hibernateProperties">
            <props>
                <prop key="hibernate.show_sql">false</prop>
                <prop key="hibernate.format_sql">true</prop>
                <prop key="hibernate.transaction.factory_class">
                  org.hibernate.transaction.JDBCTransactionFactory
                </prop>
                <prop key="hibernate.dialect">
                  org.hibernate.dialect.HSQLDialect
                </prop>
                <prop key="hibernate.connection.pool_size">0</prop>
                <prop key="hibernate.connection.driver_class">
                  org.hsqldb.jdbcDriver
                </prop>
                <prop key="hibernate.connection.url">
                  jdbc:hsqldb:data/weather;shutdown=true
                </prop>
                <prop key="hibernate.connection.username">sa</prop>
                <prop key="hibernate.connection.password"></prop>
                <prop key="hibernate.connection.autocommit">true</prop>
                <prop key="hibernate.jdbc.batch_size">0</prop>
            </props>
        </property>
    </bean>

    <bean id="locationDAO"
             class="org.sonatype.mavenbook.weather.persist.LocationDAO">
        <property name="sessionFactory" ref="sessionFactory"/>
    </bean>

    <bean id="weatherDAO"
             class="org.sonatype.mavenbook.weather.persist.WeatherDAO">
        <property name="sessionFactory" ref="sessionFactory"/>
    </bean>
 </beans>
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

在这个application context中，我们完成了一些事情。DAO从sessionFactory bean获取Hibernate Session对象。这个bean是一个AnnotationSessionFactoryBean的实例，并由一列annotatedClasses填充。注意这列标注类就是定义在我们simple-model模块中的那些类。接下来，sessionFactory通过一组Hibernate配置属性(hibernateProperties)配置。该例中，我们的Hibernate属性定义了许多设置：

hibernate.dialect
: 该设置控制如何生成数据库的SQL。由于我们正在使用HSQLDB数据库，我们的数据库方言设置成org.hibernate.dialect.HSQLDialect。Hibernate有所有主流数据库的方言，如Oracle，MySQL，Postgres和SQL Server。

hibernate.connection.*
: 该例中，我们从Spring配置中配置JDBC连接属性。我们的应用被配置成运行在./data/weather目录下的HSQLDB上。在实际的企业应用中，你更可能会使用如JNDI的东西以从你的应用程序代码中抽出数据库配置。

最后，在这个bean定义文件中，两个simple-persist DAO对象被创建并给予了对于刚定义的sessionFactory bean的引用。就像simple-weather中的Spring application context，这个applicationContext-persist.xml文件定义了一个大型企业应用设计中一个子模块的架构。如果你曾经从事过大量持久化类的集合相关的工作，你可能会发现，这些与你应用程序独立的application context文件，能帮助你快速的理解所有类之间的关系。

simple-persist中还有最后一块不清楚的地方。本章后面，我们将看到如何使用Maven Hibernate3插件，根据标注的模型对象来生成数据库schema。为了使它正确工作，Maven Hibernate3插件需要读取JDBC连接配置参数，那一列标注的类，以及src/main/resources中名为hibernate.cfg.xml文件的Hibernate配置。该文件（它重复了一些applicationContext-persist.xml中的配置）的目的是能让Maven Hibernate3插件能仅仅依靠标注就能生成数据定义语言（DDL）。如例 7.11 “simple-persist 的hibernate.cfg.xml”。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

例 7.11. simple-persist 的 hibernate.cfg.xml

```
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
  <session-factory>

    <!-- SQL dialect -->
    <property name="dialect">org.hibernate.dialect.HSQLDialect</property>

    <!-- Database connection settings -->
    <property name="connection.driver_class">org.hsqldb.jdbcDriver</property>
    <property name="connection.url">jdbc:hsqldb:data/weather</property>
    <property name="connection.username">sa</property>
    <property name="connection.password"></property>
    <property name="connection.shutdown">true</property>

    <!-- JDBC connection pool (use the built-in one) -->
    <property name="connection.pool_size">1</property>

    <!-- Enable Hibernate's automatic session context management -->
    <property name="current_session_context_class">thread</property>

    <!-- Disable the second-level cache  -->
    <property name="cache.provider_class">
      org.hibernate.cache.NoCacheProvider
    </property>

    <!-- Echo all executed SQL to stdout -->
    <property name="show_sql">true</property>

    <!-- disable batching so HSQLDB will propagate errors correctly. -->
    <property name="jdbc.batch_size">0</property>

    <!-- List all the mapping documents we're using -->
    <mapping class="org.sonatype.mavenbook.weather.model.Atmosphere"/>
    <mapping class="org.sonatype.mavenbook.weather.model.Condition"/>
    <mapping class="org.sonatype.mavenbook.weather.model.Location"/>
    <mapping class="org.sonatype.mavenbook.weather.model.Weather"/>
    <mapping class="org.sonatype.mavenbook.weather.model.Wind"/>

  </session-factory>
</hibernate-configuration>
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

例 7.10 “simple-persist 的 Spring Application Context”和例 7.1 “simple-parent 项目的 POM” 的内容是冗余的。Spring Application Context XML是被web应用和命令行应用使用的，而hibernate.cfg.xml的存在只是为了支持Maven Hibernate3插件。本章的后面，我们将会看到如何使用hibernate.cfg.xml和Maven Hibernate3插件，根据simple-model中定义的标注对象模型，来生成一个数据库schema。hibernate.cfg.xml会配置JDBC连接属性并且为Maven Hibernate3插件枚举标注模型类的列表。

7.6. simple-webapp模块

该web应用中项目simple-webapp中定义。这个简单web应用项目将会定义两个Spring MVC控制器：WeatherController和HistoryController。两者都会引用simple-weather和simple-persist中定义的组件。Spring容器在应用程序的web.xml中配置，该文件引用了simple-weather中的applicationContext-weather.xml文件和simple-persist中的applicationContext-persist.xml文件。这个简单web应用的组件架构如图 7.3 “Spring MVC 控制器引用simple-weather和simple-persist中的组件”显示。

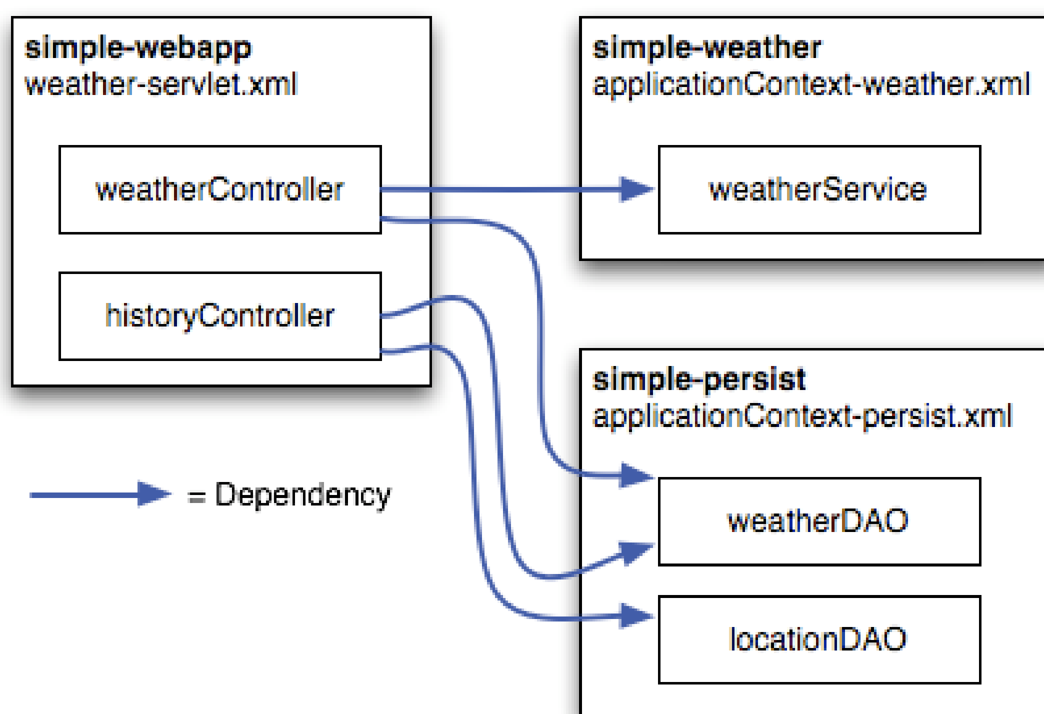


图 7.3. Spring MVC 控制器引用simple-weather和simple-persist中的组件

simple-webapp的POM如例 7.12 “simple-webapp的POM”显示。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

```
      <version>1.0</version>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring</artifactId>
      <version>2.0.7</version>
    </dependency>
    <dependency>
      <groupId>org.apache.velocity</groupId>
      <artifactId>velocity</artifactId>
      <version>1.5</version>
    </dependency>
  </dependencies>
  <build>
    <finalName>simple-webapp</finalName>
    <plugins>
      <plugin> #
        <groupId>org.mortbay.jetty</groupId>
        <artifactId>maven-jetty-plugin</artifactId>
        <dependencies>#
          <dependency>
            <groupId>hsqldb</groupId>
            <artifactId>hsqldb</artifactId>
            <version>1.8.0.7</version>
          </dependency>
        </dependencies>
      </plugin>
      <plugin>
        <groupId>org.codehaus.mojo</groupId> #
        <artifactId>hibernate3-maven-plugin</artifactId>
        <version>2.0</version>
        <configuration>
          <components>
            <component>
              <name>hbm2ddl</name>
              <implementation>annotationconfiguration</implementation> #
            </component>
          </components>
        </configuration>
        <dependencies>
          <dependency>
            <groupId>hsqldb</groupId>
            <artifactId>hsqldb</artifactId>
            <version>1.8.0.7</version>
          </dependency>
        </dependencies>
      </plugin>
    </plugins>
  </build>
</project>
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

随着书本的推进以及样例变得越来越大，你会注意到pom.xml开始呈现得有一些笨重，这里我们配置了四个依赖和两个插件。让我们详细查看一下这个POM然后详述其中一些重要的配置点：

❶ simple-webapp项目定义了四个依赖：来自于Apache Geronimo的Servlet 2.4规格说明实现，simple-weather服务类库，simple-persist持久化类库，以及整个Spring Framework 2.0.7。

❷ Maven Jetty插件以最简单的方式加入到该项目，我们只是添加一个引用了对应groupId和artifactId的plugin元素。配置这个插件如此平常意味着这个插件的开发者做了很好的工作提供了足够的默认值，在大部分情况下不需要被重写。如果你需要重写一些Jetty插件的配置，那么就需要提供configuration元素。

❸ 在我们的build配置中，我们还配置了Maven Hibernate3插件来访问内嵌的HSQLDB实例。要让Maven Hibernate3插件能成功的使用JDBC连接该数据库，该插件需要引用classpath中的HSQLDB JDBC驱动。为了让这个插件能使用该依赖，我们在plugin声明下面添加了一个dependency声明。在该例中，我们引用了hsqldb:hsqldb:1.8.0.7。这个Hibernate插件也需要JDBC驱动来创建数据库，所以我们也在它的配置中添加了这个依赖。

❹ 这个Maven Hibernate插件正是该POM变得有趣的地方。在下一节，我们将会运行hbm2ddl目标来生成HSQLDB数据库。在这个pom.xml中，我们包含了对hibernate3-maven-plugin版本2.0的引用，该插件由Codehaus Mojo维护。

❺ Maven Hibernate3插件有不同的方法获取Hibernate映射信息，这些信息适用于Hibernate3插件的不同用例。如果你正在使用Hibernate映射XML文件(.hbm.xml)，你会要使用hbm2java目标生成模型类，你会将implementation设置成configuration。如果你使用Hibernate3插件逆向工程从一个数据库产生.hbm.xml文件和模型类，你会需要一个jdbcconfiguration的implementation。在本例中，我们使用现存的标注对象模型来生成一个数据库。换句话说，我们有我们的Hibernate映射，但我们还没有数据库。在这种用例中，正确的implementation值应该是annotationconfiguration。Maven Hibernate3插件在后面的一节第　7.7　节　“运行这个Web应用"中详细讨论。

注意

一个常见的错误是使用extensions配置添加一个插件需要的依赖。这是强烈不推荐的因为extensions会在你的项目中造成classpath污染，以及其它令人讨厌的副作用。此外，extensions行为正在2.1中被重做，最后你都会要改变它。唯一的对extensions的正常使用是定义新的wagon实现。

接下来，我们将我们的注意力转移到两个处理所有请求的Spring MVC控制器。两个控制器都引用了在simple-weather和simple-persist中定义的bean。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

例 7.13. simple-webapp WeatherController

```
package org.sonatype.mavenbook.web;

import org.sonatype.mavenbook.weather.model.Weather;
import org.sonatype.mavenbook.weather.persist.WeatherDAO;
import org.sonatype.mavenbook.weather.WeatherService;
import javax.servlet.http.*;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.Controller;

public class WeatherController implements Controller {

  private WeatherService weatherService;
  private WeatherDAO weatherDAO;

  public ModelAndView handleRequest(HttpServletRequest request,
      HttpServletResponse response) throws Exception {

    String zip = request.getParameter("zip");
    Weather weather = weatherService.retrieveForecast(zip);
    weatherDAO.save(weather);
    return new ModelAndView("weather", "weather", weather);
  }

  public WeatherService getWeatherService() {
    return weatherService;
  }

  public void setWeatherService(WeatherService weatherService) {
    this.weatherService = weatherService;
  }

  public WeatherDAO getWeatherDAO() {
    return weatherDAO;
  }

  public void setWeatherDAO(WeatherDAO weatherDAO) {
    this.weatherDAO = weatherDAO;
  }
}
```

`WeatherController`实现了MVC Controller接口，该接口强制要求实现如上例中的`handleRequest()`方法。如果你看一下该方法的主要内容，你会看到它调用了`weatherService`实例变量的`retrieveForecast()`方法。不像前面的章节中，有一个

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

Servlet来初始化WeatherService类，WeatherController是一个带有weatherService属性的bean。Spring Ioc容器会负责将weatherService组件注入到控制器。同时也注意我们并没有在这个控制器实现中使用WeatherFormatter；而是将retrieveForecast()返回的Weather对象传递给了ModelAndView的构造函数。ModelAndView类将被用来呈现Velocity模板，这个模板有对${weather}变量的引用。weather.vm模板存储在src/main/webapp/WEB-INF/vm，如???所示。

在这个WeatherController中，在我们呈现天气预报输出之前，我们将WeatherService返回的Weather对象传递给WeatherDAO的save()方法。这里我们使用Hibernate将Weather对象保存到HSQLDB数据库。之后，在HistoryController中，我们将看如何能够获取由WeatherController保存的天气预报历史信息。

例 7.14. 由 WeatherController 呈现的 weather.vm 模板

```
<b>Current Weather Conditions for:
  ${weather.location.city}, ${weather.location.region},
  ${weather.location.country}</b><br/>

<ul>
  <li>Temperature: ${weather.condition.temp}</li>
  <li>Condition: ${weather.condition.text}</li>
  <li>Humidity: ${weather.atmosphere.humidity}</li>
  <li>Wind Chill: ${weather.wind.chill}</li>
  <li>Date: ${weather.date}</li>
</ul>
```

Velocity模板的语法简单易懂，变量通过${}标记引用。大括弧里面的表达式引用weather变量的一个属性，或者该变量属性的属性。weather变量是由WeatherController传递给该模板的。

HistoryController用来获取那些已经由WeatherController请求过的最近历史天气预报信息。任何时候当我们从WeatherController获取预报的时候，该控制器通过WeatherDAO将Weather对象保存至数据库。WeatherDAO然后使用Hibernate将Weather对象剖析成一组相关数据库表的记录行。HistoryController如例 7.15 “simple-web 的 HistoryController”所示。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

例 7.15. simple-web 的 HistoryController

```
package org.sonatype.mavenbook.web;

import java.util.*;
import javax.servlet.http.*;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.Controller;
import org.sonatype.mavenbook.weather.model.*;
import org.sonatype.mavenbook.weather.persist.*;

public class HistoryController implements Controller {

  private LocationDAO locationDAO;
  private WeatherDAO weatherDAO;

  public ModelAndView handleRequest(HttpServletRequest request,
      HttpServletResponse response) throws Exception {
    String zip = request.getParameter("zip");
    Location location = locationDAO.findByZip(zip);
    List<Weather> weathers = weatherDAO.recentForLocation( location );

    Map<String,Object> model = new HashMap<String,Object>();
    model.put( "location", location );
    model.put( "weathers", weathers );

    return new ModelAndView("history", model);
  }

  public WeatherDAO getWeatherDAO() {
    return weatherDAO;
  }

  public void setWeatherDAO(WeatherDAO weatherDAO) {
    this.weatherDAO = weatherDAO;
  }

  public LocationDAO getLocationDAO() {
    return locationDAO;
  }

  public void setLocationDAO(LocationDAO locationDAO) {
    this.locationDAO = locationDAO;
  }
}
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

HistoryController被注入了两个定义在simple-persist中的DAO对象。这两个DAO是HistoryController的bean属性：WeatherDAO和LocationDAO。HistoryController的目标是获取一个与zip参数对应的Weather对象列表。当WeatherDAO将Weather对象保存至数据库，它不只是保存邮编，它保存了一个与simple-model中Weather对象相关的Location对象。为了获取一个Weather对象的List，首先通过调用LocationDAO的findByZip()方法，获取与zip参数对应的Location对象。

一旦获得了Location对象，HistoryController之后就会尝试获取与给定的Location相匹配的Weather对象。在获取了List<Weather>之后，一个HashMap被创建以存储两个变量，供如???中显示的history.vm Velocity模板使用。

例 7.16. 由 HistoryController 呈现的 history.vm

```
<b>
Weather History for: ${location.city}, ${location.region}, ${location.country}
</b>
<br/>

#foreach( $weather in $weathers )
  <ul>
    <li>Temperature: $weather.condition.temp</li>
    <li>Condition: $weather.condition.text</li>
    <li>Humidity: $weather.atmosphere.humidity</li>
    <li>Wind Chill: $weather.wind.chill</li>
    <li>Date: $weather.date</li>
  </ul>
#end
```

src/main/webapp/WEB-INF/vm中的history.vm模板引用了location变量以输出天气预报位置的相关信息。该模板使用了一个Velocity的控制结构，为了循环weathers变量中的每个元素。weathers中的每个元素被赋给了一个名为weather的变量，#foreach和#end中间的模板用来呈现每个预报输出。

你已经看到了这些Controller实现，以及它们如何引用定义在simple-weather和simple-persist中的其它bean，它们相应HTTP请求，让那些知道如何呈现Velocity模板的神奇的模板系统来控制输出。所有的魔法都在位于src/main/webapp/WEB-INF/weather-servlet.xml的Spring appliction context中配置。这个XML文件配置了控制器并引用了其它Spring管理的bean，它由ServletContextListener载入，后者同时也被配置从classpath中载入了applicationContext-weather.xml和applicationContext-persist.xml。让我们仔细看一下???中展示的weather-servlet.xml。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

例 7.17. weather-servlet.xml 中的 Spring 控制器配置

```
<beans>
     <bean id="weatherController" #
           class="org.sonatype.mavenbook.web.WeatherController">
       <property name="weatherService" ref="weatherService"/>
       <property name="weatherDAO" ref="weatherDAO"/>
     </bean>

     <bean id="historyController"
           class="org.sonatype.mavenbook.web.HistoryController">
       <property name="weatherDAO" ref="weatherDAO"/>
       <property name="locationDAO" ref="locationDAO"/>
     </bean>

     <!-- you can have more than one handler defined -->
     <bean id="urlMapping"
     class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
          <property name="urlMap">
               <map>
                    <entry key="/weather.x"> #
                         <ref bean="weatherController" />
                    </entry>
                    <entry key="/history.x">
                         <ref bean="historyController" />
                    </entry>
               </map>
          </property>
     </bean>


     <bean id="velocityConfig" #
   class="org.springframework.web.servlet.view.velocity.VelocityConfigurer">
       <property name="resourceLoaderPath" value="/WEB-INF/vm/"/>
     </bean>

     <bean id="viewResolver" #
   class="org.springframework.web.servlet.view.velocity.VelocityViewResolver">
       <property name="cache" value="true"/>
       <property name="prefix" value=""/>
       <property name="suffix" value=".vm"/>
       <property name="exposeSpringMacroHelpers" value="true"/>
     </bean>
</beans>
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

❶ weather-servlet.xml定义了两个控制器作为Spring管理的bean。weatherController有两个属性，引用weatherService和weatherDAO。historyController引用了weatherDAO和locationDAO bean。当ApplicationContext被创建的时候，它所处的环境能够访问simple-persist和simple-weather中定义的ApplicationContext。在???中你将看到如何配置Spring以归并多个Spring配置文件的组件。

❷ urlMapping bean定义了调用WeatherController和HistoryController的URL模式。该例中，我们使用SimpleUrlHandlerMapping，将/weather.x映射到WeatherController，将/history.x映射到HistoryController。

❸ 由于我们正使用Velocity模板引擎，我们需要传入一些配置选项。在velocityConfig bean中，我们告诉Velocity从/WEB-INF/vm目录中寻找所有的模板。

❹ 最后，viewResolver使用VelocityViewResolver类配置。Spring中有很多viewResolver实现，从用来呈现JSP或者JSTL页面的标准viewResolver，到用来呈现Freemarker模板的viewResolver。本例中，我们配置Velocity模板引擎，设置默认的前缀和后缀，它们将被自动附加到那些传给ModelAndView的模板名前后。

最后，simple-webapp项目中有一个web.xml，提供了这个web应用的基本配置。web.xml文件如???所示：

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛


```
<web-app id="simple-webapp" version="2.4"
     xmlns="http://java.sun.com/xml/ns/j2ee"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
                         http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
  <display-name>Simple Web Application</display-name>

  <context-param> #
    <param-name>contextConfigLocation</param-name>
    <param-value>
      classpath:applicationContext-weather.xml
      classpath:applicationContext-persist.xml
    </param-value>
  </context-param>

  <context-param> #
    <param-name>log4jConfigLocation</param-name>
    <param-value>/WEB-INF/log4j.properties</param-value>
  </context-param>

  <listener> #
    <listener-class>
      org.springframework.web.util.Log4jConfigListener
    </listener-class>
  </listener>

  <listener>
    <listener-class> #
     org.springframework.web.context.ContextLoaderListener
    </listener-class>
  </listener>

  <servlet> #
    <servlet-name>weather</servlet-name>
    <servlet-class>
      org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping> #
    <servlet-name>weather</servlet-name>
    <url-pattern>*.x</url-pattern>
  </servlet-mapping>
</web-app>
```

❶ 这里有一些魔法能让我们在项目中重用applicationContext-weather.xml和applicationContext-persist.xml。contextConfigLocation由ContextLoaderListener用来创建一个ApplicationContext。当一个weather servlet被创建的时候，???中的weather-servlet.xml将由此contextConfigLocation中创建的ApplicationContext赋值。用这种方式，你可以在另外的项目中定义一组bean，然后可以通过classpath引用这些bean。由于simple-persist和simple-weather的JAR将会位于WEB-INF/lib，我们所要做的只是使用classpath:前缀来引用这些文件。（另一种选择是将所有这些文件拷贝到/WEB-INF，然后用过如/WEB-INF/applicationContext-persist.xml的方式引用它们）。

❷ log4jConfigLocation用来告诉Log4JConfigListener哪里去寻找Log4J日志配置。该例中，我们告诉Log4J在/WEB-INF/log4j.properties中寻找。

❸ 这里确保当web应用启动的时候Log4J系统被配置。将Log4JConfigListener放在ContextLoaderListener前面十分重要；否则你可能丢失那些指向阻止应用启动问题的重要日志信息。如果你有一个特别大的Spring管理的bean集合，而其中一个碰巧在应用启动的时候出问题了，应用很可能会不能启动。如果你在Spring启动之前有了日志，你就有机会看到警告或错误信息。如果你在Spring启动之前没有配置日志，你就不知道为什么你的应用不能启动了。

❹ ContextLoaderListener本质上是一个Spring容器。当应用启动的时候，这个监听器会根据contextConfigLocation参数构造一个ApplicationContext。

❺ 我们定义一个名为weather的Spring MVC DispatcherServlet。这会让Spring从/WEB-INF/weather-servlet.xml寻找Spring配置文件。你可以拥有任意多的DispatcherServlet，一个DispatcherServlet可以包含一个或多个Spring MVC Controller实现。

❻ 所有以.x结尾的请求都会被路由至weather servlet。注意.x扩展名没有任何特殊的意义，这是一个随意的选择，你可以使用任意你喜欢的URL模式。

7.7. 运行这个Web应用

为了运行这个web应用，你首先需要使用Hibernate3插件构造数据库。为此，在项目simple-webapp目录下运行如下命令：

```
$ mvn hibernate3:hbm2ddl
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'hibernate3'.
[INFO] org.codehaus.mojo: checking for updates from central
[INFO] ------------------------------------------------------------------------
[INFO] Building Chapter 7 Simple Web Application
[INFO]    task-segment: [hibernate3:hbm2ddl]
[INFO] ------------------------------------------------------------------------
[INFO] Preparing hibernate3:hbm2ddl
...
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛


```
10:24:56,151  INFO org.hibernate.tool.hbm2ddl.SchemaExport - export complete
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
```

在此之后，应该有一个/usr/local/hudson/hudson-home/jobs/maven-guide-zh-to-production/workspace/content-zh/data目录包含了HSQLDB数据库。你可以使用jetty启动web应用：

```
$ mvn jetty:run
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'jetty'.
[INFO] ------------------------------------------------------------------------
[INFO] Building Chapter 7 Simple Web Application
[INFO]    task-segment: [jetty:run]
[INFO] ------------------------------------------------------------------------
[INFO] Preparing jetty:run
...
[INFO] [jetty:run]
[INFO] Configuring Jetty for project: Chapter 7 Simple Web Application
...
[INFO] Context path = /simple-webapp
[INFO] Tmp directory =  determined at runtime
[INFO] Web defaults = org/mortbay/jetty/webapp/webdefault.xml
[INFO] Web overrides =  none
[INFO] Starting jetty 6.1.7 ...
2008-03-25 10:28:03.639::INFO:  jetty-6.1.7
...
2147 INFO  DispatcherServlet  - FrameworkServlet 'weather': \
           initialization completed in 1654 ms
2008-03-25 10:28:06.341::INFO:  Started SelectChannelConnector@0.0.0.0:8080
[INFO] Started Jetty Server
```

Jetty启动之后，你可以载入http://localhost:8080/simple-webapp/weather.x?zip=60202，然后就能在你的浏览器中看到Evanston, IL的天气。更改邮编后你就能看到你自己的天气报告了。

```
Current Weather Conditions for: Evanston, IL, US

    * Temperature: 42
    * Condition: Partly Cloudy
    * Humidity: 55
    * Wind Chill: 34
    * Date: Tue Mar 25 10:29:45 CDT 2008
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

7.8. simple-command模块

simple-command项目是simple-webapp的一个命令行版本。这个命令行工具有这同样的模块依赖：simple-persist和simple-weather。现在你需要从命令行运行这个simple-command工具，而非通过web浏览器与该应用交互。

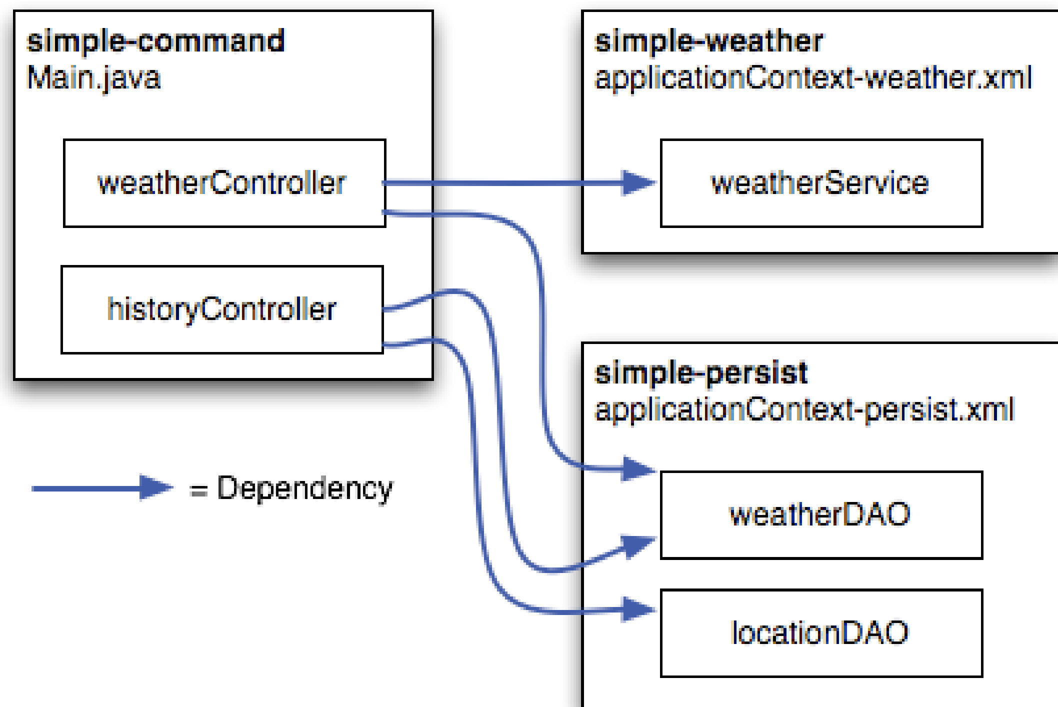


图 7.4. 引用 simple-weather 和 simple-persist 的命令行应用

```
            <descriptorRef>jar-with-dependencies</descriptorRef>
          </descriptorRefs>
        </configuration>
      </plugin>
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>hibernate3-maven-plugin</artifactId>
        <version>2.1</version>
        <configuration>
          <components>
            <component>
              <name>hbm2ddl</name>
              <implementation>annotationconfiguration</implementation>
            </component>
          </components>
        </configuration>
        <dependencies>
          <dependency>
            <groupId>hsqldb</groupId>
            <artifactId>hsqldb</artifactId>
            <version>1.8.0.7</version>
          </dependency>
        </dependencies>
      </plugin>
    </plugins>
  </build>

  <dependencies>
    <dependency>
      <groupId>org.sonatype.mavenbook.ch07</groupId>
      <artifactId>simple-weather</artifactId>
      <version>1.0</version>
    </dependency>
    <dependency>
      <groupId>org.sonatype.mavenbook.ch07</groupId>
      <artifactId>simple-persist</artifactId>
      <version>1.0</version>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring</artifactId>
      <version>2.0.7</version>
    </dependency>
    <dependency>
      <groupId>hsqldb</groupId>
      <artifactId>hsqldb</artifactId>
      <version>1.8.0.7</version>
    </dependency>
  </dependencies>
</project>
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

这个POM创建一个包含了如例 7.20 “simple-command 的 Main 类”所示的org.sonatype.mavenbook.weather.Main类的JAR文件。在这个POM中我们配置了Maven Assembly插件使用内置的名为jar-with-dependencies的装配描述符来创建一个JAR文件，该文件包含了运行应用所需要的所有二进制代码，包括项目本身的字节码以及所有依赖文件的字节码。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

```
    // Read the Zip Code from the Command-line (if none supplied use 60202)
    String zipcode = "60202";
    try {
      zipcode = args[0];
    } catch (Exception e) {
    }

    // Read the Operation from the Command-line (if none supplied use weather)
    String operation = "weather";
    try {
      operation = args[1];
    } catch (Exception e) {
    }

    // Start the program
    Main main = new Main(zipcode);

    ApplicationContext context =
      new ClassPathXmlApplicationContext(
        new String[] { "classpath:applicationContext-weather.xml",
                       "classpath:applicationContext-persist.xml" });
    main.weatherService = (WeatherService) context.getBean("weatherService");
    main.locationDAO = (LocationDAO) context.getBean("locationDAO");
    main.weatherDAO = (WeatherDAO) context.getBean("weatherDAO");
    if( operation.equals("weather")) {
      main.getWeather();
    } else {
      main.getHistory();
    }
  }

  private String zip;

  public Main(String zip) {
    this.zip = zip;
  }

  public void getWeather() throws Exception {
    Weather weather = weatherService.retrieveForecast(zip);
    weatherDAO.save( weather );
    System.out.print(new WeatherFormatter().formatWeather(weather));
  }

  public void getHistory() throws Exception {
    Location location = locationDAO.findByZip(zip);
    List<Weather> weathers = weatherDAO.recentForLocation(location);
    System.out.print(new WeatherFormatter().formatHistory(location, weathers));
  }
}
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

这个Main类有对于WeatherDAO，LocationDAO，以及 WeatherService的引用。该类的静态main()方法：

- 从第一个命令行参数读取邮编。
- 从第二个命令行参数读取操作。如果操作是“weather”，最新的天气将会从web服务获得。如果操作是“history”，该程序会从本地数据库获取历史天气记录。
- 根据来自于simple-persist和simple-weather的两个XML文件载入Spring ApplicationContext。
- 创建一个Main的实例。
- 使用来自于Spring ApplicationContext的bean填充weatherService，weatherDAO，和 locationDAO。
- 根据特定的操作运行相应的getWeather()或者getHistory()方法。

在web应用中我们使用Spring VelocityViewResolver来呈现一个Velocity模板。在这个单机实现中给我们需要编写一个简单的类来通过Velocity模板呈现天气数据。例 7.21 “WeatherFormatter 使用 Velocity 模板呈现天气数据”是WeatherFormatter的代码清单，这个类有两个方法来呈现天气报告和天气历史信息。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

例 7.21. WeatherFormatter 使用 Velocity 模板呈现天气数据

```
package org.sonatype.mavenbook.weather;

import java.io.InputStreamReader;
import java.io.Reader;
import java.io.StringWriter;
import java.util.List;

import org.apache.log4j.Logger;
import org.apache.velocity.VelocityContext;
import org.apache.velocity.app.Velocity;

import org.sonatype.mavenbook.weather.model.Location;
import org.sonatype.mavenbook.weather.model.Weather;

public class WeatherFormatter {

  private static Logger log = Logger.getLogger(WeatherFormatter.class);

  public String formatWeather( Weather weather ) throws Exception {
    log.info( "Formatting Weather Data" );
    Reader reader =
      new InputStreamReader( getClass().getClassLoader().
                                  getResourceAsStream("weather.vm"));
    VelocityContext context = new VelocityContext();
    context.put("weather", weather );
    StringWriter writer = new StringWriter();
    Velocity.evaluate(context, writer, "", reader);
    return writer.toString();
  }

  public String formatHistory( Location location, List<Weather> weathers )
        throws Exception {
    log.info( "Formatting History Data" );
    Reader reader =
      new InputStreamReader( getClass().getClassLoader().
                                  getResourceAsStream("history.vm"));
    VelocityContext context = new VelocityContext();
    context.put("location", location );
    context.put("weathers", weathers );
    StringWriter writer = new StringWriter();
    Velocity.evaluate(context, writer, "", reader);
    return writer.toString();
  }
}
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

weather.vm模板简单的打印邮编对应的城市，国家，区域以及当前的气温。history.vm模板打印位置信息并遍历存储在本地数据库中的天气预报记录。两者都位于/usr/local/hudson/hudson-home/jobs/maven-guide-zh-to-production/workspace/content-zh/src/main/resources。

例 7.22. weather.vm Velocity 模板

```
****************************************
Current Weather Conditions for:
  ${weather.location.city},
  ${weather.location.region},
  ${weather.location.country}
****************************************

 * Temperature: ${weather.condition.temp}
 * Condition: ${weather.condition.text}
 * Humidity: ${weather.atmosphere.humidity}
 * Wind Chill: ${weather.wind.chill}
 * Date: ${weather.date}
```

例 7.23. history.vm Velocity 模板

```
Weather History for:
${location.city},
${location.region},
${location.country}


#foreach( $weather in $weathers )
****************************************
 * Temperature: $weather.condition.temp
 * Condition: $weather.condition.text
 * Humidity: $weather.atmosphere.humidity
 * Wind Chill: $weather.wind.chill
 * Date: $weather.date
#end
```

7.9. 运行这个命令行程序

simple-command项目被配置成创建一个单独的包含项目本身字节码以及所有依赖字节码的JAR文件。要创建这个装配制品，从simple-command项目目录运行Maven Assembly插件的assembly目标：

```
$ mvn assembly:assembly
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛


```
[INFO] ------------------------------------------------------------------------
[INFO] Building Chapter 7 Simple Command Line Tool
[INFO]    task-segment: [assembly:assembly] (aggregator-style)
[INFO] ------------------------------------------------------------------------
[INFO] [resources:resources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:compile]
[INFO] Nothing to compile - all classes are up to date
[INFO] [resources:testResources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:testCompile]
[INFO] Nothing to compile - all classes are up to date
[INFO] [surefire:test]
...
[INFO] [jar:jar]
[INFO] Building jar: .../simple-parent/simple-command/target/simple-command.jar
[INFO] [assembly:assembly]
[INFO] Processing DependencySet (output=)
[INFO] Expanding: .../.m2/repository/.../simple-weather-1-SNAPSHOT.jar into \
                                                        /tmp/archived-file-set.9325
[INFO] Expanding: .../.m2/repository/.../simple-model-1-SNAPSHOT.jar into \
                                                        /tmp/archived-file-set.2012
[INFO] Expanding: .../.m2/repository/../hibernate-3.2.5.ga.jar into \
                                                        /tmp/archived-file-set.1296
... skipping 25 lines of dependency unpacking ...
[INFO] Expanding: .../.m2/repository/.../velocity-1.5.jar into /tmp/archived-file-s
[INFO] Expanding: .../.m2/repository/.../commons-lang-2.1.jar into \
                                                        /tmp/archived-file-set.1329
[INFO] Expanding: .../.m2/repository/.../oro-2.0.8.jar into /tmp/archived-file-set.
[INFO] Building jar: .../simple-parent/simple-command/target/simple-command-jar-wit
```

构建过程经过了生命周期中编译字节码，运行测试，然后最终为该项目构建一个JAR。然后`assembly:assembly`目标创建一个带有依赖的JAR，它将所有依赖解压到一个临时目录，然后将所有字节码收集到`target/`目录下一个名为`simple-command-jar-with-dependencies.jar`的JAR中。这个“超级的”JAR的体重为15MB。

在你运行这个命令行工具之前，你需要调用Hibernate3插件的`hbm2ddl`目标来创建HSQLDB数据库。在`simple-command`目录运行如下的命令：

```
$ mvn hibernate3:hbm2ddl
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'hibernate3'.
[INFO] org.codehaus.mojo: checking for updates from central
[INFO] ------------------------------------------------------------------------
[INFO] Building Chapter 7 Simple Command Line Tool
```

```
[INFO]    task-segment: [hibernate3:hbm2ddl]
[INFO] ------------------------------------------------------------------------
[INFO] Preparing hibernate3:hbm2ddl
...
10:24:56,151  INFO org.hibernate.tool.hbm2ddl.SchemaExport - export complete
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
```

在此之后，你应该能在simple-command下看到data/目录。该data/目录包含了HSQLDB数据库。要运行这个命令行天气预报器，在simple-command/项目目录下运行如下命令：

```
$ java -cp target/simple-command-jar-with-dependencies.jar \
           org.sonatype.mavenbook.weather.Main 60202
2321 INFO  YahooRetriever  - Retrieving Weather Data
2489 INFO  YahooParser  - Creating XML Reader
2581 INFO  YahooParser  - Parsing XML Response
2875 INFO  WeatherFormatter  - Formatting Weather Data
****************************************
Current Weather Conditions for:
  Evanston,
  IL,
  US
****************************************


 * Temperature: 75
 * Condition: Partly Cloudy
 * Humidity: 64
 * Wind Chill: 75
 * Date: Wed Aug 06 09:35:30 CDT 2008
```

要运行历史查询，运行如下命令：

```
$ java -cp target/simple-command-jar-with-dependencies.jar \
           org.sonatype.mavenbook.weather.Main 60202 history
2470 INFO  WeatherFormatter  - Formatting History Data
Weather History for:
Evanston, IL, US

****************************************
 * Temperature: 39
 * Condition: Heavy Rain
 * Humidity: 93
 * Wind Chill: 36
 * Date: 2007-12-02 13:45:27.187
****************************************
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛


```
 * Temperature: 75
 * Condition: Partly Cloudy
 * Humidity: 64
 * Wind Chill: 75
 * Date: 2008-08-06 09:24:11.725
****************************************
 * Temperature: 75
 * Condition: Partly Cloudy
 * Humidity: 64
 * Wind Chill: 75
 * Date: 2008-08-06 09:27:28.475
```

7.10. 小结

到目前为止我们花了大量时间在一些不是和Maven直接相关的话题上。这么做是为了演示一些完整的并且有意义的样例项目，以能让你用来帮助实现你的现实系统。我们并没有走任何捷径来快速生成完好的结果，也没有使用Ruby on Rails式样的东西让你感到惊讶目眩，说你可以在“简单的10分钟内！”创建并完成一个Java企业级应用。市场上这样的东西太多了，有太多的人试图卖给你最简单的框架，又只需要你投入零的时间关注。本章我们想要做的是展现给你整个的画面，整个多模块构建的生态系统。这里我们展现的Maven处于一个你能经常看到的自然应用的上下文中——不是快餐式的，10分钟的简单介绍，往Apache Ant扔烂泥，说服你采用Apache Maven。

如果你离开本章，想知道它到底和Maven有什么关系，那么我们就成功了。我们演示了一个复杂的项目集合，使用了流行的框架，并且使用声明式构建将它们绑在了一起。事实上本章60%以上的内容是在解释Spring和Hibernate，而大部分时间，Maven暂时离开了。这样是可行的。它让你集中注意力于应用本身，而非构建过程。我们专门的来讨论那些在这个人造项目中用到的技术，而不是花时间讨论Maven，以及那些为了“构建一个集成了Spring和Hibernate的构建”你必须做的工作。如果你开始使用Maven，并且花时间学习它，你肯定会开始获益，因为你不需要花时间去编写一些程式化的构建脚本。你不用再花时间去考虑构建中那些平常的方面。

你可以使用本章介绍的骨架项目作为你自己项目的基础，这么做的机会是，你会发现，你会根据需要创建越来越多的模块。例如，基于本章样例的项目可能有两个单独的模型模块，两个持久化模块以将数据持久化到不同数据库，一些web应用，以及一个Java手机应用。总的来说，这个现实系统包含了15个相关的模块。重点是，你已经看到了本书中最复杂的多模块样例，但你应该知道该样例也仅仅触及了所有Maven可能性的表面。

7.10.1. 编写接口项目程序

本章展现了一个多模块项目，该项目比第 6 章一个多模块项目中展示的简单样例复杂得多，然而它也只是现实项目的一个简化。在大型项目中，你可能发现你正构建一个如图 7.5 “编写接口项目程序”的系统。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

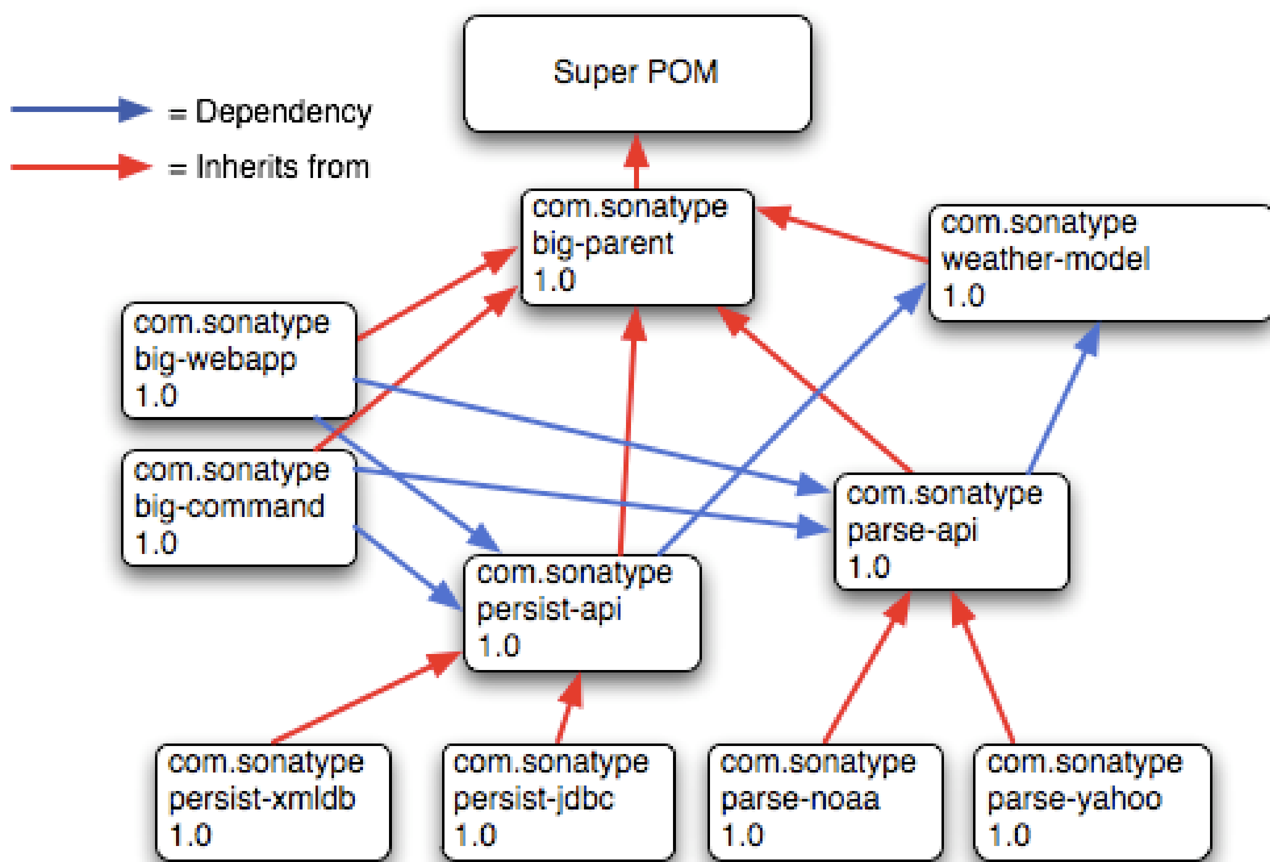


图 7.5. 编写接口项目程序

当我们使用术语接口项目的时候，我们是指一个只包含了接口和常量的Maven项目。在图 7.5 “编写接口项目程序”中，接口项目是`persist-api`和`parse-api`。如果`big-command`和`big-webapp`针对于`persist-api`中定义的接口编写，那么它就能很容易的切换到另一种持久化类库的实现。该图显示了两个`persist-api`项目的实现，一个将数据存储于XML数据库，另一个将数据存储于关系数据库。如果你使用本章中的一些概念，你就能看到如何仅仅通过传入一个标记，让程序换入一个不同的Spring application context XML文件，换出持久化实现的数据源。就像应用程序本身的OO设计一样，通常将接口Maven项目从实现Maven项目中分离是一种很明智的做法。

第 8 章 优化和重构POM

8.1. 简介

在第 7 章多模块企业级项目中，我们展示了很多Maven POM文件结合起来生成一个功能齐全的多模块构建。虽然那一章的样例想要表现一个真实的应用——与数据库交互，与web服务交互，本身又暴露两个接口：一个web应用，还有一个命令行；但这毕竟是我们杜撰出来的样例。要表现一个真实项目的复杂度可远远超出了你正阅读的本书的能力范围。真实生活中的项目往往经历了很多年的发展，由很多关注点不同的不同组的开发者维护。现实项目中，你通常会评价由其它人所做的决定或者设计。本章，我们回头看一下第 I 部分 "Maven实战"中的样例，问问我们自己，基于我们对Maven的了解，能否对该样例做更合理的优化。Maven很强大，能根据你的需要变得很简单或者很复杂。因此，通常完成同样一个任务有很多种方法，而且通常没有一个"最正确"的方式来配置你的Maven项目。

别误解最后一句话，认为它批准你可以要求Maven做一些本非它设计目的的事情。虽然Maven允许多样的方式，但当然也有"一种Maven的方式"，如此使用Maven你将能够有更高的生产效率，因为Maven本身就是被设计成这么用的。本章要做的就是传达一些能在已有项目上进行的优化手段。为什么我们不在一开始就介绍一个优化的POM呢？为教育而设计的POM与为了效率而设计的POM有着不同的需求。虽然在你的~/.m2/settings.xml中定义一些设置比在pom.xml中声明一个profile简单得多，但写书及读书主要的还是一步一步来，确保没有在你没准备的好情况下就介绍新的概念。在第 I 部分 "Maven实战"中，我们也花力气避免让太多的信息淹没读者，这样，我们就跳过了一些很重要的概念，如将在本章介绍的dependencyManagement。

在第 I 部分 "Maven实战"中有一些例子，本书的作者通过捷径掩盖了一些重要的细节，以带你关注那些特定章节的重点。你学习了如何创建一个Maven项目，不用去辛苦的读完几百页的所有原理介绍，就能编译并安装它。这么做是因为我们相信对于新的Maven用户来说，快速的呈现结果比在一个持续过长的故事中迂回更重要。在你开始使用Maven之后，你应该知道如何分析项目和的POM。本章，我们回头看看第 7 章多模块企业级项目中所留下的样例。

8.2. POM清理

优化一个多模块项目的POM最好通过几步来做，因为我们需要关注很多区域。总的来说，我们是寻找一个POM中的重复，或者多个兄弟POM中的重复。当你开始一个项目，或者项目进化得很快，有一些依赖和插件的重复是可以接受的，但随着项目的成熟以及模块的增多，你会需要花一些时间来重构共同的依赖和配置点。随着项目的变大，使你的POM更高效能很大程度的帮助你管理复杂度。不管什么时候遇到一些重复的信息片段，通常都有更好的配置方式。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

8.3. 优化依赖

如果你仔细看一下第 7 章多模块企业级项目中创建的不同POM，就会注意到几种重复模式。我们能看到的第一种模式是：一些依赖如spring和hibernate-annotations在多个模块中被声明。每个hibernate依赖都重复排除了javax.transaction。第二种重复模式是：有一些依赖是关联的，共享同样的版本。这种情况通常是因为某个项目的发布版本中包含了多个紧密关联的组件。例如，看一下依赖hibernate-annotations和hibernate-commons-annotations，两者的版本都是3.3.0.ga，而且我们可以预料这两个依赖的版本只会一起向前改变。hibernate-annotations和hibernate-commons-annotations都是JBoss发布的同一个项目的组件，当有新的版本发布的时候，两个依赖都会改变。最后的重复模式是：兄弟模块依赖和兄弟模块版本的重复。Maven提供的简单机制能让你将所有这些依赖重构到一个父POM。

就像你项目的源码一样，任何时候你在POM中有重复，你就开启了通往麻烦之路的大门。重复依赖声明使得很难保证一个大项目中的版本一致性。当你只有两个或者三个模块的时候，可能这不是一个大问题，但当你的组织正使用一个大型的，多模块Maven构建来管理分布于很多部门的数百个组件，一个依赖间的版本不匹配能够造成混乱。项目中一个对于名为ASM的字节码操作包的依赖版本不一致，即使处于项目层次的三层以下，如果该模块被依赖，还是可以影响到由另一个完全不同的开发组维护的web应用。单元测试会通过因为它们是基于一个版本的依赖运行的，但产品可能会灾难性的失败，原因是包（比如这里是war）里存在有不同版本的类库。如果你拥有数十个项目使用比如Hibernate这样的东西，每个项目重复那些依赖和排除配置，那么有人搞坏构建的平均发生时间就会很短。由于你的Maven项目变得很复杂，依赖列表也会增大，你需要在父POM中巩固版本和依赖声明。

兄弟模块版本的重复可以造成一个特殊的令人讨厌的问题，该问题不是直接由Maven造成的，只有在你多次遇到这个问题之后你才会有认识。如果你使用Maven Release插件来运行你的发布，所有这些兄弟依赖版本都会被自动更新，因此维护它们就不是什么问题。如果simple-web版本1.3-SNAPSHOT依赖于simple-persist版本1.3-SNAPSHOT，并且你正执行一次对于两个项目的版本1.3发布，Maven Release插件很聪明，能够自动更改整个多模块项目中的所有POM。使用Realse插件来运行发布能够自动将你构建的所有版本增加到1.4-SNAPSHOT，并且release插件会将代码变更提交至代码库。发布一个大型的多模块项目会变得更简单，直到……

当开发人员将更改合并到POM中并影响了一个正进行的版本发布的时候，问题就产生了。通常一个开发人员合并更改并偶然的错误处理了对于兄弟依赖的冲突，不注意的回退到了前一个发布的版本。由于同一个依赖的连续版本通常是相互兼容的，当开发人员构建的时候，问题不会出现，甚至暂时在持续构建系统也不会发现。想像一下一个十分复杂的构建，主干上都是1.4-SNAPSHOT的组件，现在有一个开发人员A，将项目层次深处的组件A更新至依赖于组件B的1.3-SNAPSHOT版本。虽然大部分开发者都使用1.4-SNAPSHOT了，如果组件B的1.3-SNAPSHOT和1.4-SNAPSHOT相互兼容的话，该构建还是会成

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

功。Maven会继续使用从开发者本地仓库获取的组件B的1.3-SNAPSHOT版本构建该项目。所有事情开起来都很流畅，项目构建，持续集成构建都没问题，有人可能有一个关于组件B的诡异的bug，我们也暂时将其认为是一个小问题记下来，然后继续下面的事情。

有个人，让我们称其为马虎先生，在组件A中有一个合并冲突，然后错误的将组件A对于组件B的依赖设为了1.3-SNAPSHOT，而项目的其它部分继续向前推进。一堆开发人员试图修复组件B的bug，诡异的是他们在产品环境中看不到bug被修复了。偶然间有人看了下组件A然后意识到这个依赖指向了一个错误的版本。幸运的是，这个bug还没有大到要消耗很多钱或时间，但是马虎先生感到自己十分愚蠢，由于这次的兄弟依赖关系混乱问题，人们也没以前那么信任他了。（还好，马虎先生意识到这是一个用户行为错误而非Maven的错，但可能它会写一个糟糕的博客去无休止的抱怨Maven来使自己好受一点。）

幸运的是，只要你做一些微小的更改，依赖重复和兄弟依赖不匹配问题就能简单的预防。我们要做的第一件事是找出所有被用于一个以上模块的依赖，然后将其向上移到父POM的dependencyManagement片段。我们先不管兄弟依赖。simple-parent的POM现在包含内容如下：

```
<project>
  ...
  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring</artifactId>
        <version>2.0.7</version>
      </dependency>
      <dependency>
        <groupId>org.apache.velocity</groupId>
        <artifactId>velocity</artifactId>
        <version>1.5</version>
      </dependency>
      <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-annotations</artifactId>
        <version>3.3.0.ga</version>
      </dependency>
      <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-commons-annotations</artifactId>
        <version>3.3.0.ga</version>
      </dependency>
      <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate</artifactId>
        <version>3.2.5.ga</version>
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛


```
          <exclusions>
            <exclusion>
              <groupId>javax.transaction</groupId>
              <artifactId>jta</artifactId>
            </exclusion>
          </exclusions>
        </dependency>
      </dependencies>
    </dependencyManagement>
    ...
  </project>
```

在这些依赖配置被上移之后，我们需要为每个POM移除这些依赖的版本，否则它们会覆盖定义在父项目中的dependencyManagement。这里我只是简单展示一下simple-model：

```
<project>
  ...
  <dependencies>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-annotations</artifactId>
    </dependency>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate</artifactId>
    </dependency>
  </dependencies>
  ...
</project>
```

下一步我们应该做的是修复hibernate-annotations和hibernate-commons-annotations的版本重复问题，因为这两个版本应该是一致的，我们通过创建一个称为hibernate-annotations-version的属性。结果simple-parent的片段看起来这样：

```
<project>
  ...
  <properties>
    <hibernate.annotations.version>3.3.0.ga</hibernate.annotations.version>
  </properties>

  <dependencyManagement>
    ...
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-annotations</artifactId>
      <version>${hibernate.annotations.version}</version>
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛


```
    </dependency>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-commons-annotations</artifactId>
      <version>${hibernate.annotations.version}</version>
    </dependency>
    ...
  </dependencyManagement>
  ...
</project
```

我们需要处理的最后一个问题是兄弟依赖。一种方案是像其它依赖一样将它们移到父项目的dependencyManagement中，在最顶层的父项目中定义所有兄弟项目的版本。这样做当然是可以的，但我们也可以使用内建属性org.sonatype.mavenbook和0.6-SNAPSHOT来解决这个版本问题。由于它们是兄弟依赖，在父项目中枚举它们也不能获得更多的价值，因此我们依赖于内置的0.6-SNAPSHOT属性。由于我们都共享一个共同的组，因此，通过使用内置的org.sonatype.mavenbook属性引用当前POM的组，我们能够提前保证这些声明是正确的。simple-command依赖片段现在变成了这样：

```
<project>
  ...
  <dependencies>
    ...
    <dependency>
      <groupId>org.sonatype.mavenbook</groupId>
      <artifactId>simple-weather</artifactId>
      <version>0.6-SNAPSHOT</version>
    </dependency>
    <dependency>
      <groupId>org.sonatype.mavenbook</groupId>
      <artifactId>simple-persist</artifactId>
      <version>0.6-SNAPSHOT</version>
    </dependency>
    ...
  </dependencies>
  ...
</project>
```

总结一下我们为了降低依赖重复而完成的两项优化：

上移共同的依赖至dependencyManagement
: 如果多于一个项目依赖于一个特定的依赖，你可以在dependencyManagement中列出这个依赖。父POM包含一个版本和一组排除配置，所有的子POM需要使用groupId和artifactId引用这个依赖。如果依赖已经在dependencyManagement中列出，子项目可以忽略版本和排除配置。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

为兄弟项目使用内置的项目version和groupId

使用{project.version}和org.sonatype.mavenbook来引用兄弟项目。兄弟项目基本上一直共享同样的groupId，也基本上一直共享同样的发布版本。使用0.6-SNAPSHOT可以帮你避免前面提到的兄弟版本不一致问题。

8.4. 优化插件

如果我们看一下不同的插件配置，就能看到HSQLDB依赖在很多地方有重复。不幸的是，dependencyManagement不适用于插件依赖，但我们仍然可以使用属性来巩固版本。大部分复杂的Maven多模块项目倾向于在顶层POM中定义所有的版本。这个顶层POM就成了影响整个项目的更改焦点。将版本号看成是Java类中的字符串字面量，如果你经常重复一个字面量，你可能会将它定义为一个变量，当它需要变化的时候，你就只需要在一个地方更改它。将HSQLDB的版本集中到顶层的POM中，就产生了如下的属性元素。

```
<project>
  ...
  <properties>
    <hibernate.annotations.version>3.3.0.ga</hibernate.annotations.version>
    <hsqldb.version>1.8.0.7</hsqldb.version>
  </properties>
  ...
</project>
```

下一件我们要注意的事情是hibernate3-maven-plugin配置在simple-webapp和simple-command模块中重复了。我们可以在顶层POM中管理这个插件配置，就像我们在顶层POM中使用dependencyManagement片段管理依赖一样。为此，我们要使用元素顶层POM build元素下的pluginManagement元素。

```
<project>
  ...
  <build>
    <pluginManagement>
      <plugins>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-compiler-plugin</artifactId>
          <configuration>
            <source>1.5</source>
            <target>1.5</target>
          </configuration>
        </plugin>
        <plugin>
          <groupId>org.codehaus.mojo</groupId>
          <artifactId>hibernate3-maven-plugin</artifactId>
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛


```
          <version>2.1</version>
          <configuration>
            <components>
              <component>
                <name>hbm2ddl</name>
                <implementation>annotationconfiguration</implementation>
              </component>
            </components>
          </configuration>
          <dependencies>
            <dependency>
              <groupId>hsqldb</groupId>
              <artifactId>hsqldb</artifactId>
              <version>${hsqldb.version}</version>
            </dependency>
          </dependencies>
        </plugin>
      </plugins>
    </pluginManagement>
  </build>
  ...
</project>
```

8.5. 使用Maven Dependency插件进行优化

在大型的项目中，随着依赖数目的增加，一些额外的依赖会悄悄进入项目的POM中。而当依赖改变的时候，你又常常会留下一些不再被使用的依赖，而又有时候，你会忘记显示声明你需要的类库依赖。由于Maven 2.x会在编译范围引入传递性依赖，你的项目可能编译没问题，但在产品阶段不能运行。考虑这种情况，当一个项目使用一些被广泛使用的类库如Jakarta Commons Beanutils。你没有显式的声明对Beanutils的依赖，你的项目依赖于一个项目如Hibernate，而后者有对Beanutils的传递性依赖。你的项目可能编译成功并很好的运行，但当你将Hibernate升级到一个新版本，而它不再依赖于Beantuils，你就会遇到编译和运行错误了，这种情况直到项目不能编译才能显现。同时，由于你没有显式的列出依赖的版本，Maven不能帮你解析可能出现的版本冲突问题。

一个好的经验方法是，总是为你代码引用的类显式声明依赖。如果你要引入Commons Beanutils类，你应该声明一个对于Commons Beanutils的直接依赖。幸运的是，通过字节码分析，Maven Dependency插件能够帮助你发现对于依赖的直接引用。使用我们之前优化过的新的POM，让我们看一下是否会有错误突然出现。

```
$ mvn dependency:analyze
[INFO] Scanning for projects...
[INFO] Reactor build order:
```

```
[INFO]   Chapter 8 Simple Parent Project
[INFO]   Chapter 8 Simple Object Model
[INFO]   Chapter 8 Simple Weather API
[INFO]   Chapter 8 Simple Persistence API
[INFO]   Chapter 8 Simple Command Line Tool
[INFO]   Chapter 8 Simple Web Application
[INFO]   Chapter 8 Parent Project
[INFO] Searching repository for plugin with prefix: 'dependency'.

...

[INFO] ------------------------------------------------------------------------
[INFO] Building Chapter 8 Simple Object Model
[INFO]    task-segment: [dependency:analyze]
[INFO] ------------------------------------------------------------------------
[INFO] Preparing dependency:analyze
[INFO] [resources:resources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:compile]
[INFO] Nothing to compile - all classes are up to date
[INFO] [resources:testResources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:testCompile]
[INFO] Nothing to compile - all classes are up to date
[INFO] [dependency:analyze]
[WARNING] Used undeclared dependencies found:
[WARNING]    javax.persistence:persistence-api:jar:1.0:compile
[WARNING] Unused declared dependencies found:
[WARNING]    org.hibernate:hibernate-annotations:jar:3.3.0.ga:compile
[WARNING]    org.hibernate:hibernate:jar:3.2.5.ga:compile
[WARNING]    junit:junit:jar:3.8.1:test

...

[INFO] ------------------------------------------------------------------------
[INFO] Building Chapter 8 Simple Web Application
[INFO]    task-segment: [dependency:analyze]
[INFO] ------------------------------------------------------------------------
[INFO] Preparing dependency:analyze
[INFO] [resources:resources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:compile]
[INFO] Nothing to compile - all classes are up to date
[INFO] [resources:testResources]
[INFO] Using default encoding to copy filtered resources.
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛


```
[INFO] [compiler:testCompile]
[INFO] No sources to compile
[INFO] [dependency:analyze]
[WARNING] Used undeclared dependencies found:
[WARNING]    org.sonatype.mavenbook.ch08:simple-model:jar:1.0:compile
[WARNING] Unused declared dependencies found:
[WARNING]    org.apache.velocity:velocity:jar:1.5:compile
[WARNING]    javax.servlet:jstl:jar:1.1.2:compile
[WARNING]    taglibs:standard:jar:1.1.2:compile
[WARNING]    junit:junit:jar:3.8.1:test
```

在上面截取的输出中，我们能看到运行dependency:analyze目标的结果。该目标分析这个项目，查看是否有直接依赖，或者一些引用了但不是直接声明的依赖。在simple-model项目中，dependency插件指出有一个对于javax.persistence:persistence-api的“使用了但为未声明的依赖”。为了进一步调查，到simple-model目录下运行dependency:tree目标，该目标会列出项目中所有的直接和传递性依赖。

```
$ mvn dependency:tree
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'dependency'.
[INFO] ------------------------------------------------------------------------
[INFO] Building Chapter 8 Simple Object Model
[INFO]    task-segment: [dependency:tree]
[INFO] ------------------------------------------------------------------------
[INFO] [dependency:tree]
[INFO] org.sonatype.mavenbook.ch08:simple-model:jar:1.0
[INFO] +- org.hibernate:hibernate-annotations:jar:3.3.0.ga:compile
[INFO] |  \- javax.persistence:persistence-api:jar:1.0:compile
[INFO] +- org.hibernate:hibernate:jar:3.2.5.ga:compile
[INFO] |  +- net.sf.ehcache:ehcache:jar:1.2.3:compile
[INFO] |  +- commons-logging:commons-logging:jar:1.0.4:compile
[INFO] |  +- asm:asm-attrs:jar:1.5.3:compile
[INFO] |  +- dom4j:dom4j:jar:1.6.1:compile
[INFO] |  +- antlr:antlr:jar:2.7.6:compile
[INFO] |  +- cglib:cglib:jar:2.1_3:compile
[INFO] |  +- asm:asm:jar:1.5.3:compile
[INFO] |  \- commons-collections:commons-collections:jar:2.1.1:compile
[INFO] \- junit:junit:jar:3.8.1:test
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
```

从上面的输出我们可以看到persistence-api依赖来自于hibernate。对该模块源码的粗略扫描会展现很多的javax.persistence的import语句，这让我们确信直接引用了这个依赖。简单的修复手段是添加对这个依赖的直接引用。该例中，我们将依赖版本放

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

到simple-parent的dependencyManagement片段中，因为这个依赖链接到Hibernate，而Hiberate的版本是在这里声明的。最终你会想要升级的项目的HIbernate版本，在Hibernate依赖旁边列出persistence-api依赖的版本能让你在将来升级父POM中的Hibernate版本的时候，更明显的看到两者关联。

如果你查看simple-web模块的dependency:analyze输出，你会看到这里我们也需要添加对simple-model的直接依赖。simple-webapp中的代码直接引用simple-model中的模型对象，而现在simple-model是通过simple-persist的传递性依赖暴露给simple-webapp的。既然兄弟依赖共享同样的version和groupId，因此这个依赖可以在simple-webapp的pom.xml中用org.sonatype.mavenbook和0.6-SNAPSHOT定义。

Maven Dependency插件是如何发现这些问题的呢？dependency:analyze如何知道什么类和依赖是你项目的字节码直接引用的？Dependency插件使用ObjectWeb ASM[1]工具包来分析字节码。Dependency插件使用ASM来遍历当前项目中的所有类，构建一个所有其它被引用的类的列表。之后它遍历所有的依赖，直接依赖和传递性依赖，然后标记所有在直接依赖中发现的类。任何没有在直接依赖中找到的类会在传递性依赖中被发现，然后，“使用的，但未声明的依赖”列表就产生了。

相反的，未使用的，但声明的依赖列表就相对比较难验证了，而且该列表没有“使用的，但未声明的依赖”有用。一种情况，一些依赖只在运行时或测试时使用，它们不会在字节码中被发现。你能在输出中很明显的看到它们的存在，例如，JUnit就在这个列表中，但是它是需要的，因为它被用来做单元测试。你也会在simple-web模块中注意到Velocity和Servlet API依赖出现在这个列表中，它们也是需要的，因为，虽然项目的类中没有任何对这些依赖的直接引用，但在运行的时候它们是必要的。

小心移除那些未使用，但声明的依赖，除非你拥有很好的测试覆盖率，否则你很可能引入了一个运行时错误。字节码优化还会突然出现更险恶的问题，例如，编译器可以合法的替换常量的值，优化并移除引用。此时移除依赖会造成编译错误，但是工具却分析显示该依赖未被使用。将来的Maven Dependency插件版本会提供个更好的技术来检测或忽略这些类型的问题。

你应该定期的使用dependency:analyze工具来检测你项目中的这些普遍错误。你可以配置它，当一些条件被发现的时候，让构建失败，它也能够用来生成报告。

8.6. 最终的POM

作为一个最后的总结，最终的POM文件被列出以作为本章的参考。这是simple-parent的顶层POM.

[1] http://asm.objectweb.org/

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

```
  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring</artifactId>
        <version>2.0.7</version>
      </dependency>
      <dependency>
        <groupId>org.apache.velocity</groupId>
        <artifactId>velocity</artifactId>
        <version>1.5</version>
      </dependency>
      <dependency>
        <groupId>javax.persistence</groupId>
        <artifactId>persistence-api</artifactId>
        <version>1.0</version>
      </dependency>
      <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-annotations</artifactId>
        <version>${hibernate.annotations.version}</version>
      </dependency>
      <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-commons-annotations</artifactId>
        <version>${hibernate.annotations.version}</version>
      </dependency>
      <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate</artifactId>
        <version>3.2.5.ga</version>
        <exclusions>
          <exclusion>
            <groupId>javax.transaction</groupId>
            <artifactId>jta</artifactId>
          </exclusion>
        </exclusions>
      </dependency>
    </dependencies>
  </dependencyManagement>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

下面是该工具的命令行版本，`simple-command`的最终POM。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

```
          <artifactId>maven-jar-plugin</artifactId>
          <configuration>
            <archive>
              <manifest>
                <mainClass>org.sonatype.mavenbook.weather.Main</mainClass>
                <addClasspath>true</addClasspath>
              </manifest>
            </archive>
          </configuration>
        </plugin>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-surefire-plugin</artifactId>
          <configuration>
            <testFailureIgnore>true</testFailureIgnore>
          </configuration>
        </plugin>
        <plugin>
         <artifactId>maven-assembly-plugin</artifactId>
          <configuration>
            <descriptorRefs>
              <descriptorRef>jar-with-dependencies</descriptorRef>
            </descriptorRefs>
          </configuration>
        </plugin>
      </plugins>
    </pluginManagement>
  </build>

  <dependencies>
    <dependency>
      <groupId>org.sonatype.mavenbook</groupId>
      <artifactId>simple-weather</artifactId>
      <version>0.6-SNAPSHOT</version>
    </dependency>
    <dependency>
      <groupId>org.sonatype.mavenbook</groupId>
      <artifactId>simple-persist</artifactId>
      <version>0.6-SNAPSHOT</version>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring</artifactId>
    </dependency>
    <dependency>
      <groupId>org.apache.velocity</groupId>
      <artifactId>velocity</artifactId>
    </dependency>
  </dependencies>
</project>
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

接下来是simple-model项目的POM。simple-model包含了所有应用的模型对象。

例 8.3. simple-model 的最终 POM

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                             http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.sonatype.mavenbook.ch08</groupId>
    <artifactId>simple-parent</artifactId>
    <version>1.0</version>
  </parent>
  <artifactId>simple-model</artifactId>
  <packaging>jar</packaging>

  <name>Chapter 8 Simple Object Model</name>

  <dependencies>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-annotations</artifactId>
    </dependency>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate</artifactId>
    </dependency>
    <dependency>
      <groupId>javax.persistence</groupId>
      <artifactId>persistence-api</artifactId>
    </dependency>
  </dependencies>
</project>
```

下一个POM是simple-persist项目的POM。simple-persist项目包含了使用Hibernate实现的所有持久化逻辑。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

例 8.4. simple-persist 的最终 POM

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                             http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.sonatype.mavenbook.ch08</groupId>
    <artifactId>simple-parent</artifactId>
    <version>1.0</version>
  </parent>
  <artifactId>simple-persist</artifactId>
  <packaging>jar</packaging>

  <name>Chapter 8 Simple Persistence API</name>

  <dependencies>
    <dependency>
      <groupId>org.sonatype.mavenbook</groupId>
      <artifactId>simple-model</artifactId>
      <version>0.6-SNAPSHOT</version>
    </dependency>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate</artifactId>
    </dependency>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-annotations</artifactId>
    </dependency>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-commons-annotations</artifactId>
    </dependency>
    <dependency>
      <groupId>org.apache.geronimo.specs</groupId>
      <artifactId>geronimo-jta_1.1_spec</artifactId>
      <version>1.1</version>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring</artifactId>
    </dependency>
  </dependencies>
</project>
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

simple-weather项目包含了解析Yahoo! Weather RSS信息源的所有逻辑。该项目依赖于simple-model项目。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

例 8.5. simple-weather 的最终 POM

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                             http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.sonatype.mavenbook.ch08</groupId>
    <artifactId>simple-parent</artifactId>
    <version>1.0</version>
  </parent>
  <artifactId>simple-weather</artifactId>
  <packaging>jar</packaging>

  <name>Chapter 8 Simple Weather API</name>

  <dependencies>
    <dependency>
      <groupId>org.sonatype.mavenbook</groupId>
      <artifactId>simple-model</artifactId>
      <version>0.6-SNAPSHOT</version>
    </dependency>
    <dependency>
      <groupId>log4j</groupId>
      <artifactId>log4j</artifactId>
      <version>1.2.14</version>
    </dependency>
    <dependency>
      <groupId>dom4j</groupId>
      <artifactId>dom4j</artifactId>
      <version>1.6.1</version>
    </dependency>
    <dependency>
      <groupId>jaxen</groupId>
      <artifactId>jaxen</artifactId>
      <version>1.1.1</version>
    </dependency>
    <dependency>
      <groupId>org.apache.commons</groupId>
      <artifactId>commons-io</artifactId>
      <version>1.3.2</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

最后，`simple-webapp`项目与`simple-weather`项目生成的类库交互，同时也将获得的天气预报数据存储至HSQLDB数据库。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

```
      <artifactId>simple-model</artifactId>
      <version>0.6-SNAPSHOT</version>
    </dependency>
    <dependency>
      <groupId>org.sonatype.mavenbook</groupId>
      <artifactId>simple-weather</artifactId>
      <version>0.6-SNAPSHOT</version>
    </dependency>
    <dependency>
      <groupId>org.sonatype.mavenbook</groupId>
      <artifactId>simple-persist</artifactId>
      <version>0.6-SNAPSHOT</version>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring</artifactId>
    </dependency>
    <dependency>
      <groupId>javax.servlet</groupId>
      <artifactId>jstl</artifactId>
      <version>1.1.2</version>
    </dependency>
    <dependency>
      <groupId>taglibs</groupId>
      <artifactId>standard</artifactId>
      <version>1.1.2</version>
    </dependency>
    <dependency>
      <groupId>org.apache.velocity</groupId>
      <artifactId>velocity</artifactId>
    </dependency>
  </dependencies>
  <build>
    <finalName>simple-webapp</finalName>
    <plugins>
      <plugin>
        <groupId>org.mortbay.jetty</groupId>
        <artifactId>maven-jetty-plugin</artifactId>
        <version>6.1.9</version>
        <dependencies>
          <dependency>
            <groupId>hsqldb</groupId>
            <artifactId>hsqldb</artifactId>
            <version>${hsqldb.version}</version>
          </dependency>
        </dependencies>
      </plugin>
    </plugins>
  </build>
</project>
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

8.7. 小结

本章为你展现了一些改进控制你项目依赖和插件的技术，以使你未来的构建维护变得轻松。我们推荐定期的用这种方式复查你的构建以确保重复以及相应的潜在问题能最小化。随着项目的成熟，新的依赖难免被引入，你可能发现之前在一个地方使用的一个依赖现在在10个地方使用了，需要将它上移。使用和未使用的依赖列表不停的在变化，它们也能够使用Maven Dependency插件轻松的清理。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

部分 II. Maven参考

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

第 9 章 项目对象模型

9.1. 简介

本章讨论Maven的核心概念——项目对象模型。在POM中，项目的坐标和结构被声明，构建被配置，与其它项目的关联也被定义。pom.xml文件定义了一个Maven项目。

9.2. POM

Maven项目，依赖，构建配置，以及构件：所有这些都是要建模和表述的对象。这些对象通过一个名为项目对象模型(Project Object Model, POM)的XML文件描述。这个POM告诉Maven它正处理什么类型的项目，如何修改默认的行为来从源码生成输出。同样的方式，一个Java Web应用有一个web.xml文件来描述，配置，及自定义该应用，一个Maven项目则通过一个 pom.xml文件定义。该文件是Maven中一个项目的描述性陈述；也是当Maven构建项目的时候需要理解的一份“地图”。

你可以将pom.xml看成是类似于Makefile或者Ant中的build.xml。当你使用GNU make来构建诸如MySQL软件的时候，你通常会有一个名为Makefile的文件，它包含了显式的指令来清理，编译，打包以及部署一个应用。在这一点上，Make，Ant，和Maven是相似的，它们都依赖于一个统一命名的文件如Makefile，build.xml，或者pom.xml，但相似的地方也仅此而已。如果你看一下Maven的pom.xml，POM的主要内容是处理描述信息：哪里存放源代码？哪里存放资源文件？打包方式是什么？如果你看一下Ant的build.xml文件，你会看到完全不同的东西。那里有显式的指令来执行一些任务，如编译一组Java类。Maven的POM是声明性的，虽然你可以通过Maven Ant插件来引入一些过程式的自定义指令，但大部分时间里，你不需要去了解项目构建的过程细节。

POM也不只是仅仅针对于构建Java项目。虽然本书的大部分样例都是Java应用，但是在Maven的项目对象模型定义中没有任何Java特定的东西。虽然Maven的默认插件是从一组源码，测试，和资源来构建一个JAR文件。但你同样可以为一个包含C#源码，使用微软工具处理一些微软私有的二进制文件的项目来定义一个POM。类似的，你也可以为一本技术书籍定义一个POM。事实上，本书的源码和本书的样例正是用一个Maven多模块项目组织的，我们使用一个Maven Docbook插件，将标准的Docbook XSL应用到一系列章节的XML文件上。还有人编写了Maven插件来将Adobe Flex代码构建成SWC和SWF，也还有人使用了Maven来构建C编写的项目。

我们已经确定了POM是描述性和声明性的，它不像Ant或者Make那样提供显式的指令，我们也注意到POM的概念不是特定于Java的。让我们深入更多的细节，看一下图 9.1 “项目对象模型”，纵览一下POM的内容。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

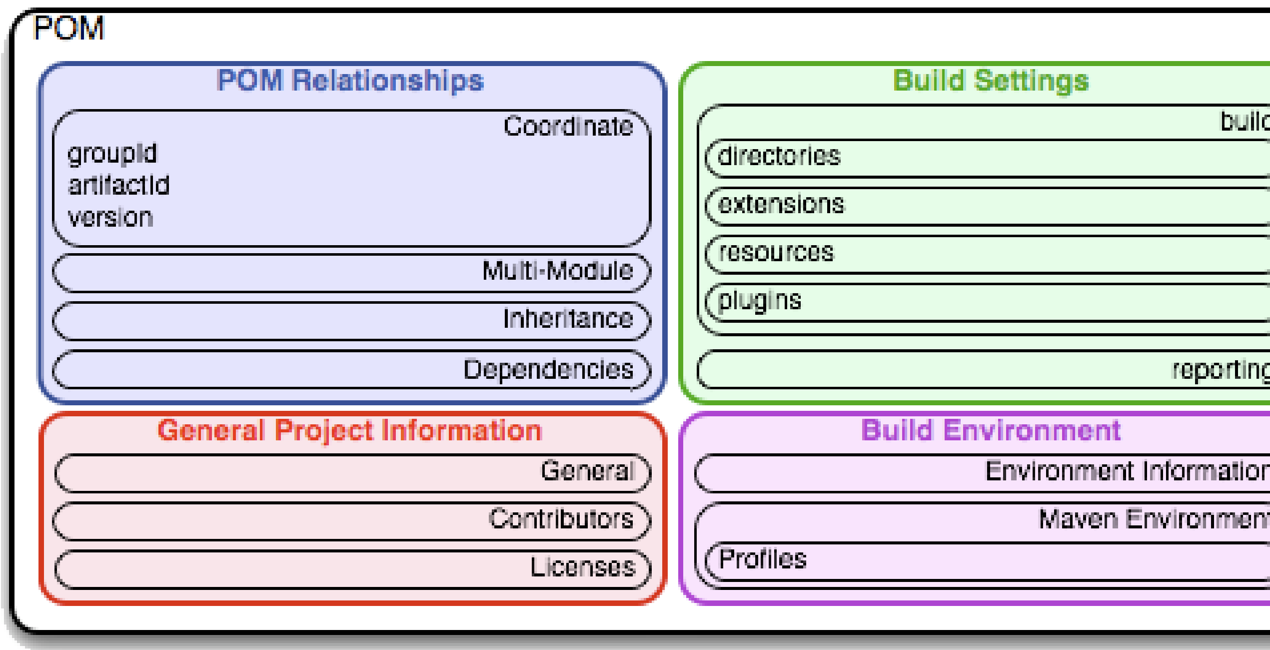


图 9.1. 项目对象模型

POM包含了四类描述和配置：

项目总体信息
: 它包含了一个项目的而名称，项目的URL，发起组织，以及项目的开发者贡献者列表和许可证。

构建设置
: 在这一部分，我们自定义Maven构建的默认行为。我们可以更改源码和测试代码的位置，可以添加新的插件，可以将插件目标绑定到生命周期，我们还可以自定义站点生成参数。

构建环境
: 构建环境包含了一些能在不同使用环境中 激活的profile。例如，在开发过程中你可能会想要将应用部署到一个而开发服务器上，而在产品环境中你会需要将应用部署到产品服务器上。构建环境为特定的环境定制了构建设置，通常它还会由~/.m2中的自定义settings.xml补充。这个settings文件将会在第 11 章构建Profile中，以及第 A.1 节 “简介”中的附录 A,附录：Settings细节小节中讨论。

POM关系
: 一个项目很少孤立存在；它会依赖于其它项目，可能从父项目继承POM设置，它要定义自身的坐标，可能还会包含子模块。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

9.2.1. 超级POM

在深入钻研一些样例POM之前，让我们先快速看一下超级POM。所有的Maven项目的POM都扩展自超级POM。超级POM定义了一组被所有项目共享的默认设置。它是Maven安装的一部分，可以在/usr/local/maven/lib中的maven-2.0.9-uber.jar文件中找到。如果你看一下这个JAR文件，你会看到在包org.apache.maven.project下看到一个名为pom-4.0.0.xml的文件。这个Maven的超级POM如例 9.1 “超级POM”所示。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

```
        <artifactId>maven-install-plugin</artifactId>
        <version>2.2</version>
      </plugin>
      <plugin>
        <artifactId>maven-jar-plugin</artifactId>
        <version>2.2</version>
      </plugin>
      <plugin>
        <artifactId>maven-javadoc-plugin</artifactId>
        <version>2.4</version>
      </plugin>
      <plugin>
        <artifactId>maven-plugin-plugin</artifactId>
        <version>2.3</version>
      </plugin>
      <plugin>
        <artifactId>maven-rar-plugin</artifactId>
        <version>2.2</version>
      </plugin>
      <plugin>
        <artifactId>maven-release-plugin</artifactId>
        <version>2.0-beta-7</version>
      </plugin>
      <plugin>
        <artifactId>maven-resources-plugin</artifactId>
        <version>2.2</version>
      </plugin>
      <plugin>
        <artifactId>maven-site-plugin</artifactId>
        <version>2.0-beta-6</version>
      </plugin>
      <plugin>
        <artifactId>maven-source-plugin</artifactId>
        <version>2.0.4</version>
      </plugin>
      <plugin>
         <artifactId>maven-surefire-plugin</artifactId>
         <version>2.4.2</version>
      </plugin>
      <plugin>
        <artifactId>maven-war-plugin</artifactId>
        <version>2.1-alpha-1</version>
      </plugin>
    </plugins>
  </pluginManagement>

  <reporting>
    <outputDirectory>target/site</outputDirectory>
  </reporting>
</project>
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

这个超级POM定义了一些由所有项目继承的标准配置变量。对这些变量的简单解释如下：

❶ 默认的超级POM定义了一个单独的远程Maven仓库，ID为`central`。这是所有Maven客户端默认配置访问的中央Maven仓库。该配置可以通过一个自定义的`settings.xml`文件来覆盖。注意这个默认的超级POM关闭了从中央Maven仓库下载snapshot构件的功能。如果你需要使用一个snapshot仓库，你就要在你的`pom.xml`或者`settings.xml`中自定义仓库设置。Settings和profile将会在第 11 章构建Profile中和第 A.1 节 “简介”中的附录 A, 附录: Settings细节小节中具体介绍。

❷ 中央Maven仓库同时也包含Maven插件。默认的插件仓库就是这个中央仓库。Snapshot被关闭了，而且更新策略被设置成了“从不”，这意味着Maven将永远不会自动更新一个插件，即使新版本的插件发布了。

❸ `build`元素设置Maven标准目录布局中那些目录的默认值。

❹ 从Maven 2.0.9开始，超级POM为核心插件提供了默认版本。这么做是为那些没有在它们POM中指定插件版本的用户提供一些稳定性。

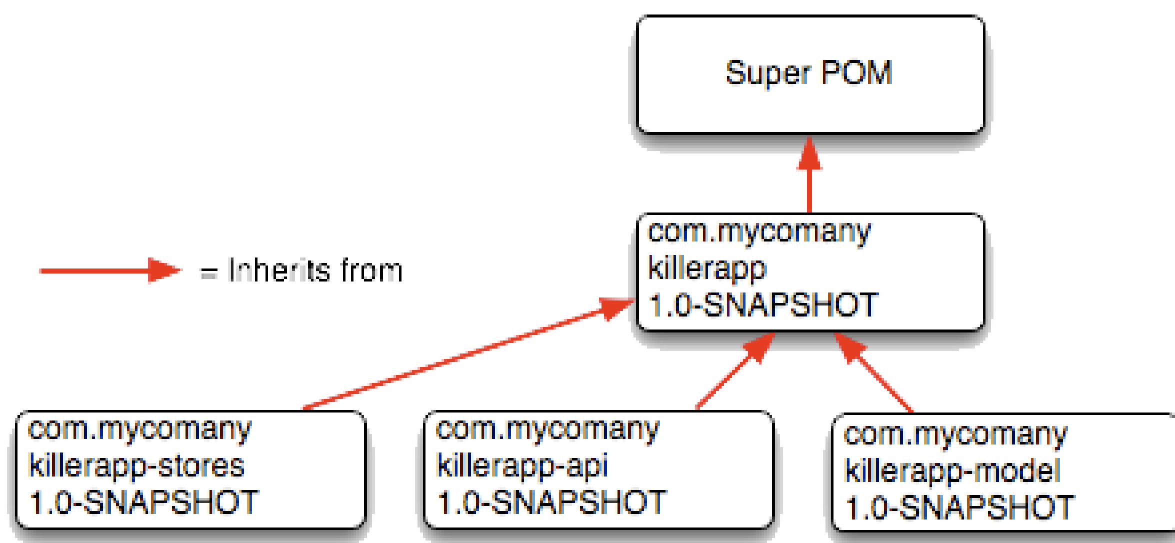


图 9.2. 超级POM永远是最基础的父POM

9.2.2. 最简单的POM

所有的Maven POM都继承自超级POM（在前面的小节第 9.2.1 节 “超级POM”中介绍）。如果你只是编写一个简单的项目，从`src/main/java`目录的源码生成一个JAR，想要运行`src/test/java`中的JUnit测试，想要使用mvn site构建一个项目站点，你不需要自定义任何东西。在这种情况下，你所需要的，是如例 9.2 “最简单的POM”所示的一

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

个最简单的POM。这个POM定义了groupId，artifactId和version：这三项是所有项目都需要的坐标。

例 9.2. 最简单的POM

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.sonatype.mavenbook.ch08</groupId>
  <artifactId>simplest-project</artifactId>
  <version>1</version>
</project>
```

对一个简单的项目来说，这样一个简单的POM已经足够了——例如，一个生成单个JAR文件的Java类库。它不需要和任何其它项目关联，没有任何依赖，也缺少基本的信息如名字和URL。如果创建了这个文件，然后创建了一个子目录src/main/java，并且放入了一些源代码，运行mvn package将会生成一个名为target/simple-project-1.jar的JAR文件。

9.2.3. 有效POM

最简单的POM能带给我们“有效POM”的概念。由于POM可以从其它POM继承配置，你就需要一直考虑超级POM，再加上任何父POM，以及最后当前项目的POM这些所有配置的结合。Maven开始于超级POM，然后使用一个或多个父POM覆盖默认配置，最后使用当前项目的POM来覆盖之前生成的配置结果。最后你得到了一个混合了各个POM配置的有效POM。如果你想要查看项目的有效POM，你需要运行Maven Help插件的effective-pom目标，该插件已经之前在小节第 2.7 节 “使用Maven Help插件”中介绍。在pom.xml文件所在的目录执行以下的命令以运行effective-pom目标：

```
$ mvn help:effective-pom
```

执行effective-pom目标应该会打印出一个XML文档，该文档的内容是超级POM和例 9.2 “最简单的POM”中POM内容的合并。

9.2.4. 真正的POM

这里我们就不再设计一组POM，一步步的来看了，你可以自己查看一下第 I 部分 “Maven实战”中的样例。Maven就像是变色龙，你可以挑选你想要使用的特性。对于一些开源项目来说，列出开发者和贡献者，生成整洁的项目文档，使用Maven Release插件来自动管理版本发布可能很有价值。但另一方面，一些在大公司环境下的小型团队中工作的人可能对Maven的分发管理功能或者开发成员列表功能不感兴趣。本章的剩余部分将会单独的讨论POM的特性。我们不会用数十页的一组相关POM来炮轰你，而会集中于为POM中的特定的小节创建优良的参考内容。本章，我们也会讨论POM之间的关系，但不会在这里展示这样一个项目。如果你想要看这样一个示例，请参考第 7 章多模块企业级项目。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

9.3. POM语法

Maven项目中的POM永远都是基础目录下的一个名为pom.xml的文件。这个XML文档可以以XML声明开头，或者你也可以选择忽略它。所有的POM的值都通过XML元素的形式体现。

9.3.1. 项目版本

一个Maven项目发布版本号用version编码，用来分组和排序发布。Maven中的版本包含了以下部分：主版本，次版本，增量版本，和限定版本号。一个版本中，这些部分对应如下的格式：

```
<major version>.<minor version>.<incremental version>-<qualifier>
```

例如：版本“1.3.5”由一个主版本1，一个次版本3，和一个增量版本5。而一个版本“5”只有主版本5，没有次版本和增量版本。限定版本用来标识里程碑构建：alpha和beta发布，限定版本通过连字符与主版本，次版本或增量版本隔离。例如，版本“1.3-beta-01”有一个主版本1，次版本3，和一个限定版本“beta-01”。

当你想要在你的POM中使用版本界限的时候，保持你的版本号与标准一致十分重要。在第 9.4.3 节 “依赖版本界限中介绍的版本界限，允许你声明一个带有版本界限的依赖，只有你遵循标准的时候该功能才被支持。因为Maven根据本节中介绍的版本号格式来对版本进行排序。

如果你的版本号与格式<主版本>.<次版本>.<增量版本>-<限定版本>相匹配，它就能被正确的比较；“1.2.3”将被评价成是一个比“1.0.2”更新的构件，这种比较基于主版本，次版本，和增量版本的数值。如果你的版本发布号没有符合本节介绍的标准，那么你的版本号只会根据字符串被比较；“1.0.1b”和“1.2.0b”会使用字符串比较。

9.3.1.1. 版本构建号

我们还需要对版本号的限定版本进行排序。以版本号“1.2.3-alpha-2”和“1.2.3-alpha-10”为例，这里“alpha-2”对应了第二次alpha构建，而“alpha-10”对应了第十次alpha构建。虽然“alpha-10”应该被认为是比“alpha-2”更新的构建，但Maven排序的结果是“alpha-10”比“alpha-2”更旧，问题的原因就是我们刚才讨论的Maven处理版本号的方式。

Maven会将限定版本后面的数字认作一个构建版本。换句话说，这里限定版本是“alpha”，而构建版本是2。虽然Maven被设计成将构建版本和限定版本分离，但目前这种解析还是失效的。因此，“alpha-2”和“alpha-10”是使用字符串进行比较的，而根据字母和数字“alpha-10”在“alpha-2”前面。要避开这种限制，你需要对你的限定版本使用一些技巧。如果你使用“alpha-02”和“alpha-10”，这个问题就消除了，一旦Maven能正确的解析版本构建号之后，这种工作方式也还是能用。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

9.3.1.2. SNAPSHOT版本

Maven版本可以包含一个字符串字面量来表示项目正处于活动的开发状态。如果一个版本包含字符串“SNAPSHOT”，Maven就会在安装或发布这个组件的时候将该符号展开为一个日期和时间值，转换为UTC（协调世界时）。例如，如果你的项目有个版本为“1.0-SNAPSHOT”并且你将这个项目的构件部署到了一个Maven仓库，如果你在UTC2008年2月7号下午11:08部署了这个版本，Maven就会将这个版本展开成“1.0-20080207-230803-1”。换句话说，当你发布一个snapshot，你没有发布一个软件模块，你只是发布了一个特定时间的快照版本。

那么为什么要使用这种方式呢？SNAPSHOT版本在项目活动的开发过程中使用。如果你的项目依赖的一个组件正处于开发过程中，你可以依赖于一个SNAPSHOT版本，在你运行构建的时候Maven会定期的从仓库下载最新的snapshot。类似的，如果你系统的下一个发布版本是“1.4”你的项目需要拥有一个“1.4-SNAPSHOT”的版本，之后它被正式发布。

作为一个默认设置，Maven不会从远程仓库检查SNAPSHOT版本，要依赖于SNAPSHOT版本，用户必须在POM中使用`repository`和`pluginRepository`元素显式的开启下载snapshot的功能。

当发布一个项目的时候，你需要解析所有对SNAPSHOT版本的依赖至正式发布的版本。如果一个项目依赖于SNAPSHOT，那么这个依赖很不稳定，它随时可能变化。发布到非snapshot的Maven仓库（如http://repo1.maven.org/maven2）的构件不能依赖于任何SNAPSHOT版本，因为Maven的超级POM对于中央仓库关闭了snapshot。SNAPSHOT版本只用于开发过程。

9.3.1.3. LATEST 和 RELEASE 版本

当你依赖于一个插件或一个依赖，你可以使用特殊的版本值LATEST或者RELEASE。LATEST是指某个特定构件最新的发布版或者快照版(snapshot)，最近被部署到某个特定仓库的构件。RELEASE是指仓库中最后的一个非快照版本。总得来说，设计软件去依赖于一个构件的不明确的版本，并不是一个好的实践。如果你处于软件开发过程中，你可能想要使用RELEASE或者LATEST，这么做十分方便，你也不用为每次一个第三方类库新版本的发布而去更新你配置的版本号。但当你发布软件的时候，你总是应该确定你的项目依赖于某个特定的版本，以减少构建的不确定性，免得被其它不受你控制的软件版本影响。如果无论如何你都要使用LATEST和RELEASE，那么要小心使用。

Maven 2.0.9之后，Maven在超级POM中锁住了一些通用及核心Maven插件的版本号，以将某个特定版本Maven的核心Maven插件组标准化。这个变化在Maven 2.0.9中被引入，为Maven构建带来了稳定性和重现性。在Maven 2.0.9之前，Maven会自动将核心插件更新至LATEST版本。这种行为导致了很多奇怪现象，因为新版本的插件可能会有一些bug，甚至是行为变更，这往往使得原来的构建失败。当Maven自动更新核心插件的时候，我们就不能保证构建的重现性，因为插件随时都可能从中央仓库更新至一个新的版本。从

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

Maven 2.0.9开始，Maven从根本上锁住了一组核心插件的版本。非核心插件，或者说没有在超级POM中指定版本的插件仍然会使用LATEST版本去从仓库获取构件。由于这个原因，你在构件中使用任何一个自定义非核心插件的时候，都应该显式的指定版本号。

9.3.2. 属性引用

一个POM可以通过一对大括弧和前面一个美元符号来包含 对属性的引用。例如，考虑如下的POM：

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.sonatype.mavenbook</groupId>
  <artifactId>project-a</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  <build>
    <finalName>org.sonatype.mavenbook-content-zh</finalName>
  </build>
</project>
```

如果你将这段XML放入`pom.xml`，然后运行mvn help:effective-pom，你会看到输出包含这一行：

```
...
<finalName>org.sonatype.mavenbook-project-a</finalName>
...
```

在Maven读取一个POM的时候，它会在载入POM XML的时候替换这些属性的引用。在Maven的高级使用中Maven属性经常出现，这些属性和其它系统中的属性如Ant或者Velocity类似。它们是一些由`MavenProject: org.sonatype.mavenbook:content-zh:0.6-SNAPSHOT @ /usr/local/hudson/hudson-home/jobs/maven-guide-zh-to-production/workspace/content-zh/pom.xml`划界的变量。Maven提供了三个隐式的变量，可以用来访问环境变量，POM信息，和Maven Settings：

env

: `env`变量 暴露了你操作系统或者shell的环境变量。例如，在Maven POM中一个对`/usr/lib/jvm/java-1.6.0-openjdk-1.6.0.0/bin:/usr/local/bin:/usr/local/maven/bin:/usr/kerberos/sbin:/usr/kerberos/bin:/usr/java/latest/bin:/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin:/root/bin:/usr/bin:/usr/local/bin`的引用将会被`/usr/local/bin:/usr/local/maven/bin:/usr/kerberos/sbin:/usr/kerberos/bin:/usr/java/latest/bin:/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin:/root/bin:/usr/bin:/usr/local/bin`环境变量替换（或者Windows中的`%PATH%`）。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

project
: project变量暴露了POM。你可以使用点标记（.）的路径来引用POM元素的值。例如，在本节中我们使用过groupId和artifactId来设置构建配置中的finalName元素。这个属性引用的语法是：org.sonatype.mavenbook-${project.artifactId}。

settings
: settings变量暴露了Maven settings信息。可以使用点标记（.）的路径来引用settings.xml文件中元素的值。例如，${settings.offline}会引用~/.m2/settings.xml文件中offline元素的值。

注意

你可能在老的构建中看到使用${pom.xxx}或者仅仅${xxx}来引用POM属性。这些方法已被弃用，我们只应该使用${project.xxx}。

除了这三个隐式的变量，你还可以引用系统属性，以及任何在Maven POM中和构建profile中自定义的属性组。

Java系统属性
: 所有可以通过java.lang.System中getProperties()方法访问的属性都被暴露成POM属性。一些系统属性的例子是：hudson，/home/hudson，/usr/lib/jvm/java-1.6.0-openjdk-1.6.0.0/jre，和Linux。一个完整的系统属性列表可以在java.lang.System类的Javadoc中找到。

x
: 我们还可以通过pom.xml或者settings.xml中的properties元素设置自己的属性，或者还可以使用外部载入的文件中属性。如果你在pom.xml中设置了一个名为fooBar的属性，该属性就可以通过${fooBar}引用。当你构建一个系统，它针对不同的部署环境过滤资源，那么自定义属性就变得十分有用。这里是在POM中设置${foo}=bar的语法：

```
<properties>
  <foo>bar</foo>
</properties>
```

要了解更复杂的可用属性列表，查看第 13 章属性和资源过滤。

9.4. 项目依赖

Maven可以管理内部和外部依赖。一个Java项目的外部依赖可能是如Plexus，Spring Framework，或者Log4J的类库。一个内部的依赖就像在“一个简单的web应用”中描述的那样，web项目依赖于另外一个包含服务类，模型类，或者持久化逻辑的项目。例 9.3 “项目依赖”展示了一些项目依赖的例子。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

例　9.3.　项目依赖

```
<project>
  ...
  <dependencies>
    <dependency>
      <groupId>org.codehaus.xfire</groupId>
      <artifactId>xfire-java5</artifactId>
      <version>1.2.5</version>
    </dependency>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.apache.geronimo.specs</groupId>
      <artifactId>geronimo-servlet_2.4_spec</artifactId>
      <version>1.0</version>
      <scope>provided</scope>
    </dependency>
  </dependencies>
  ...
</project>
```

这里的第一个依赖是对于来自Codehaus的XFire SOAP库的编译范围（compile）依赖。如果你的项目在编译，测试，和运行中都依赖于一个类库，你就要使用这种依赖。第二种依赖是一个对于JUnit测试范围（test）的依赖。当你只有在测试的时候才引用类库的时候，你就要使用测试范围依赖。例　9.3 “项目依赖”中最后一个依赖是对于由Apache Geronimo项目实现的Servlet 2.4 API的依赖。最后一项依赖的范围是已提供的（provided）依赖。当你的开发过程只有在编译和测试时需要一个类库，而该类库在运行的时候由容器提供，那么你就需要使用已提供范围的依赖。

9.4.1.　依赖范围

例　9.3 “项目依赖”简要介绍了三种依赖范围：compile，test，和provided。范围控制哪些依赖在哪些classpath中可用，哪些依赖包含在一个应用中。让我们详细看一下每一种范围：

compile（编译范围）

compile是默认的范围；如果没有提供一个范围，那该依赖的范围就是编译范围。编译范围依赖在所有的classpath中可用，同时它们也会被打包。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

provided（已提供范围）

provided依赖只有在当JDK或者一个容器已提供该依赖之后才使用。例如，如果你开发了一个web应用，你可能在编译classpath中需要可用的Servlet API来编译一个servlet，但是你不会想要在打包好的WAR中包含这个Servlet API；这个Servlet API JAR由你的应用服务器或者servlet容器提供。已提供范围的依赖在编译classpath（不是运行时）可用。它们不是传递性的，也不会被打包。

runtime（运行时范围）

runtime依赖在运行和测试系统的时候需要，但在编译的时候不需要。比如，你可能在编译的时候只需要JDBC API JAR，而只有在运行的时候才需要JDBC驱动实现。

test（测试范围）

test范围依赖 在一般的 编译和运行时都不需要，它们只有在测试编译和测试运行阶段可用。测试范围依赖在之前的???中介绍过。

system（系统范围）

system范围依赖与provided类似，但是你必须显式的提供一个对于本地系统中JAR文件的路径。这么做是为了允许基于本地对象编译，而这些对象是系统类库的一部分。这样的构件应该是一直可用的，Maven也不会在仓库中去寻找它。如果你将一个依赖范围设置成系统范围，你必须同时提供一个systemPath元素。注意该范围是不推荐使用的（你应该一直尽量去从公共或定制的Maven仓库中引用依赖）。

9.4.2. 可选依赖

假定你正在开发一个类库 ，该类库提供高速缓存行为。你想要使用一些已存在的能够提供文件系统快速缓存和分布式快速缓存的类库，而非从空白开始写自己的快速缓存系统。再假定你想要能让最终用户选择使用文件系统高速缓存或者内存分布式高速缓存。为了缓存文件系统，你会要使用免费的类库如EHCache（http://ehcache.sourceforge.net/），为了分布式内存缓存，你想要使用免费的类库如SwarmCache（http://swarmcache.sourceforge.net/）。你将编写一个接口，并且创建一个可以被配置成使用EHCache或者SwarmCache的类库，但是你想要避免为所有依赖于你类库的项目添加全部这两个缓存类库的的依赖。

换句话说，编译这个项目的时候你需要两个依赖类库，但是你不希望在使用你类库的项目中，这两个依赖类库同时作为传递性运行时依赖出现。你可以使用如例 9.4 “声明可选依赖”中的可选依赖来完成这个任务。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

例 9.4. 声明可选依赖

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.sonatype.mavenbook</groupId>
  <artifactId>my-project</artifactId>
  <version>1.0.0</version>
  <dependencies>
    <dependency>
      <groupId>net.sf.ehcache</groupId>
      <artifactId>ehcache</artifactId>
      <version>1.4.1</version>
      <optional>true</optional>
    </dependency>
    <dependency>
      <groupId>swarmcache</groupId>
      <artifactId>swarmcache</artifactId>
      <version>1.0RC2</version>
      <optional>true</optional>
    </dependency>
    <dependency>
      <groupId>log4j</groupId>
      <artifactId>log4j</artifactId>
      <version>1.2.13</version>
    </dependency>
  </dependencies>
</project>
```

在你将这些依赖声明为可选之后，你就需要在依赖于my-project的项目中显式的引用对应的依赖。例如，如果你正编写一个应用，它依赖于my-project，并且想要使用EHCache实现，你就需要在你项目添加如下的dependency元素。

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.sonatype.mavenbook</groupId>
  <artifactId>my-application</artifactId>
  <version>1.0.0</version>
  <dependencies>
    <dependency>
      <groupId>org.sonatype.mavenbook</groupId>
      <artifactId>my-project</artifactId>
      <version>1.0.0</version>
    </dependency>
    <dependency>
      <groupId>net.sf.ehcache</groupId>
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛


```
      <artifactId>swarmcache</artifactId>
      <version>1.4.1</version>
    </dependency>
  </dependencies>
</project>
```

在理想的世界中，你不需要使用可选依赖。你可以将EHCache相关的代码放到my-project-ehcache子模块中，将SwarmCache相关的代码放到my-project-swarmcache子模块中，而非创建一个带有一系列可选依赖的大项目。这样，其它项目就可以只引用特定实现的项目，发挥传递性依赖的功效，而不用去引用my-project项目，再自己声明特定的依赖。

9.4.3. 依赖版本界限

你并不是必须为依赖声明某个特定的版本，你可以指定一个满足给定依赖的版本界限。例如，你可以指定你的项目依赖于JUnit的3.8或以上版本，或者说依赖于JUnit 1.2.10和1.2.14之间的某个版本。你可以使用如下的字符来围绕一个或多个版本号，来实现版本界限。

(,)
: 不包含量词

[,]
: 包含量词

例如，如果你想要访问JUnit任意的大于等于3.8但小于4.0的版本，你的依赖可以如例 9.5 “指定一个依赖界限：JUnit 3.8 - JUnit 4.0”编写：

例 9.5. 指定一个依赖界限：JUnit 3.8 - JUnit 4.0

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>[3.8,4.0)</version>
  <scope>test</scope>
</dependency>
```

如果想要依赖JUnit任意的不大于3.8.1的版本，你可以只指定一个上包含边界，如例 9.6 “指定一个依赖界限：JUnit <= 3.8.1”所示：

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

例 9.6. 指定一个依赖界限：JUnit <= 3.8.1

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>[,3.8.1]</version>ex-de
  <scope>test</scope>
</dependency>
```

在逗号前面或者后面的版本不是必须的，这种空缺意味着正无穷或者负无穷。例如，“[4.0,)”意思是任何大于等于4.0的版本，“(,2.0)”意思是任意小于2.0的版本。“[1.2]”意思是只有版本1.2，没有其它。

注意

当声明一个“正常的”版本如JUnit 3.8.2，内部它其实被表述成“允许任何版本，但最好是3.8.2”。意思是当侦测到版本冲突的时候，Maven会使用冲突算法来选择最好的版本。如果你指定[3.8.2]，它意味只有3.8.2会被使用，没有其它。如果其它什么地方有一个版本指定了[3.8.1]，你会得到一个构建失败报告，告诉你有版本冲突。我指出这一点是要让你知道有这一选项，但要保守的使用它，只有在确实需要的时候才使用。更好的做法是通过dependencyManagement来解决冲突。

9.4.4. 传递性依赖

一个传递性依赖就是对于一个依赖的依赖。如果project-a依赖于project-b，而后者接着依赖于project-c，那么project-c就被认为是project-a的传递性依赖。如果project-c依赖于project-d，那么project-d就也被认为是project-a的传递性依赖。Maven的部分吸引力是由于它能够管理传递性依赖，并且能够帮助开发者屏蔽掉跟踪所有编译期和运行期依赖的细节。你可以只依赖于一些包如Spring Framework，而不用担心Spring Framework的所有依赖，Maven帮你自动管理了，你不用自己去详细了解配置。

Maven是怎样完成这件事情的呢？它建立一个依赖图，并且处理一些可能发生的冲突和重叠。例如，如果Maven看到有两个项目依赖于同样的groupId和artifactId，它会自动整理出使用哪个依赖，选择那个最新版本的依赖。虽然这听起来很方便，但在一些边界情况中，传递性依赖会造成一些配置问题。在这种情况下，你可以使用依赖排除。

9.4.4.1. 传递性依赖和范围

第 9.4.1 节 “依赖范围”中提到的每种依赖范围不仅仅影响声明项目中的依赖范围，它也对所传递性依赖起作用。表达该信息最简单的方式是通过一张表来表述，如表 9.1 “范围如何影响传递性依赖”。最顶层一行代表了传递性依赖的范围。最左边的一列代

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

表了直接依赖的范围。行与列的交叉就是为某个传递性依赖指定的范围。表中的空格意思是该传递性依赖被忽略。

表 9.1. 范围如何影响传递性依赖

直接范围	传递性范围			
	compile	provided	runtime	test
compile	compile	-	runtime	-
provided	provided	provided	provided	-
runtime	runtime	-	runtime	-
test	test	-	test	-

要阐明传递性依赖范围到直接依赖范围的关系，考虑如下例子。如果project-a包含一个对于project-b的测试范围依赖，后者包含一个对于project-c的编译范围依赖。project-c将会是project-a的测试范围传递性依赖。

你可以将这看成是一个作用于依赖范围上的传递性边界。那些已提供范围和测试范围的传递性依赖往往不对项目产生影响。该规则的例外是已提供范围传递性依赖到已提供范围直接依赖还是项目的一个已提供范围依赖。编译范围和运行时范围的传递性依赖通常会影响那个一个项目，无论它的直接依赖范围是什么。编译范围的传递性依赖将会和直接依赖产生与后者相同范围的的结果。运行时范围的传递性依赖也会和直接依赖产生与后者相同范围的结果，除非当直接依赖是编译范围的时候，结果是运行时范围。

9.4.5. 冲突解决

有很多时候你需要排除一个传递性依赖，比如当你依赖于一个项目，后者又继而依赖于另外一个项目，但你的希望是，要么整个的排除这个传递性依赖，要么用另外一个提供同样功能的依赖来替代这个传递性依赖。例 9.7 “排除一个传递性依赖”展示的例子中添加了一个对于project-a的依赖，但排除了传递性依赖project-b。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

例 9.7. 排除一个传递性依赖

```
<dependency>
  <groupId>org.sonatype.mavenbook</groupId>
  <artifactId>project-a</artifactId>
  <version>1.0</version>
  <exclusions>
    <exclusion>
      <groupId>org.sonatype.mavenbook</groupId>
      <artifactId>project-b</artifactId>
    </exclusion>
  </exclusions>
</dependency>
```

通常，你会想要使用另外一个实现来替代一个传递性依赖。比如，如果你正依赖于一个类库，该类库又依赖于Sun JTA API，你会想要替换这个传递性依赖。Hibernate是一个例子。Hibernate依赖于Sun JTA API，而后者在中央Maven仓库中不可用，因为它是不能免费分发的。幸运的是，Apache Gernoimo项目创建了一些可以免费分发的独立实现类库。为了用另外的依赖来替换这个传递性依赖，你需要排除这个传递性以依赖，然后在你的项目中再声明一个依赖。例 9.8 “排除并替换一个传递性依赖”展示了这样一个替换的样例。

例 9.8. 排除并替换一个传递性依赖

```
<dependencies>
  <dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate</artifactId>
    <version>3.2.5.ga</version>
    <exclusions>
      <exclusion>
        <groupId>javax.transaction</groupId>
        <artifactId>jta</artifactId>
      </exclusion>
    </exclusions>
  </dependency>
  <dependency>
    <groupId>org.apache.geronimo.specs</groupId>
    <artifactId>geronimo-jta_1.1_spec</artifactId>
    <version>1.1</version>
  </dependency>
</dependencies>
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

在例 9.8 “排除并替换一个传递性依赖”中，没有什么标记说依赖geronimo-jta_1.1_spec是一个替换，它只是正好提供了和原来的JTA依赖一样的API。以下列举了一些你可能想要排除或者替换传递性依赖的情况：

1. 构建的groupId和artifactId已经更改了，而当前的项目需要一个与传递性依赖不同名称的版本——结果是classpath中出现了同样项目的两份内容。一般来说Maven会捕捉到这种冲突并且使用该项目的一个单独的版本，但是当artifactId和artifactId不一样的时候，Maven就会认为它们是两种不同的类库。

2. 某个构件没有在你的项目中被使用，而且该传递性依赖没有被标志为可选依赖。在这种情况下，你可能想要排除这种依赖，因为它不是你的系统需要的东西，你要尽量减少应用程序分发时的类库数目。

3. 一个构件已经在运行时的容器中提供了，因此不应该被包含在你的构件中。该情况的一个例子是，如果一个依赖依赖于如Servlet API的东西，并且你又要确保这样的依赖没有包含在web应用的WEB-INF/lib目录中。

4. 为了排除一个可能是多个实现的API的依赖。这种情况在例 9.8 “排除并替换一个传递性依赖”中阐述；有一个Sun API，需要点击许可证，并且需要耗时的手工安装到自定义仓库，对于同样的API有可免费分发版本，在中央Maven仓库中可用（Geronimo’s JTA 实现）。

9.4.6. 依赖管理

当你在你的超级复杂的企业中采用Maven之后，你有了两百多个相互关联的Maven项目，你开始想知道是否有一个更好的方法来处理依赖版本。如果每一个使用如MySQL数据库驱动依赖的项目都需要独立的列出该依赖的版本，在你需要升级到一个新版本的时候你就会遇到问题。由于这些版本号分散在你的项目树中，你需要手工的编写每一个引用该依赖的pom.xml，确保每个地方的版本号都更改了。即使使用了find，xargs，和，awk，你仍然有漏掉一个POM的风险。

幸运的是，Maven在dependencyManagement元素中为你提供了一种方式来统一依赖版本号。你经常会在一个组织或者项目的最顶层的父POM中看到dependencyManagement元素。使用pom.xml中的dependencyManagement元素能让你在子项目中引用一个依赖而不用显式的列出版本号。Maven会沿着父子层次向上走，直到找到一个拥有dependencyManagement元素的项目，然后它就会使用在这个dependencyManagement元素中指定的版本号。

例如，如果你有一大组项目使用MySQL Java connector版本5.1.2，你可以在你的多模块项目的顶层POM中定义如下的dependencyManagement元素。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

例 9.9. 在一个项目POM中定义依赖版本

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.sonatype.mavenbook</groupId>
  <artifactId>a-parent</artifactId>
  <version>1.0.0</version>
  ...
  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>5.1.2</version>
      </dependency>
      ...
    <dependencies>
  </dependencyManagement>
```

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.sonatype.mavenbook</groupId>
    <artifactId>a-parent</artifactId>
    <version>1.0.0</version>
  </parent>
  <artifactId>project-a</artifactId>
  ...
  <dependencies>
    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
    </dependency>
  </dependencies>
</project>
```

你应该注意到该子项目没有显式的列出mysql-connector-java依赖的版本。由于这个依赖在顶层POM的dependencyManagement元素中定义了，该版本号就会传播到所有子项目的mysql-connector-java依赖中。注意如果子项目定义了一个版本，它将覆盖顶层POM的dependencyManagement元素中的版本。那就是：只有在子项目没有直接声明一个版本的时候，dependencyManagement定义的版本才会被使用。

顶层POM中的依赖管理与在一个广泛共享的父POM中定义一个依赖是不同的。对初学者来说，所有依赖都会被继承。如果mysql-connector-java在顶层父项目中被作为一个依赖列出，这个层次中的所有项目都将引用该依赖。为了不添加一些不必要的依赖，使用dependencyManagement能让你统一并集中化依赖版本的管理，而不用添加那些会被所有子项目继承的依赖。换句话说，dependencyManagement元素和一个环境变量一样，能让你在一个项目下面的任何地方声明一个依赖而不用指定一个版本号。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

9.5. 项目关系

使用Maven的引人注目的原因之一是它使得追踪依赖（以及依赖的依赖）的过程非常容易。当一个项目依赖于另一个项目生成的构件，我们就说这个构件是一个依赖。在Java项目的情况下，这可以简单到比如一个项目依赖与外部的如Log4J或JUnit依赖。依赖可以为外部依赖建模，也可以管理一组相关项目的依赖，如果project-a依赖于project-b，Maven就能够很聪明的知道project-b必须在project-a之前构建。

项目关系不仅仅是依赖以及解决一个项目需要能构建出一个构件。Maven可以建模的关系还包括，某个项目是父项目，某个项目是子模块。本节给你项目中各种关系的概览，并且告诉你如何配置这些关系。

9.5.1. 坐标详解

坐标为一个项目定义一个唯一的位置，它们首先在第 3 章一个简单的Maven项目中介绍过。项目使用Maven坐标与其它项目关联。一个项目不是简单的依赖于另一个项目，而是一个带有groupId，artifactId，和version的项目依赖于另一个带有groupId，artifactId，和version的项目。回顾一下，Maven坐标有三部分组成：

groupId
: 一个groupId归类了一组相关的构件。组定义符基本上类似于一个Java包名。例如：groupId org.apache.maven是所有由Apache Maven项目生成的构件的基本groupId。组定义符在Maven仓库中被翻译成路径，例如，groupId org.apache.maven可以在repo1.maven.org[1]的/maven2/org/apache/maven目录下找到。

artifactId
: artifactId是项目的主要定义符。当你生成一个构件，这个构件将由artifactId命名。当你引用一个项目，你就需要使用artifactId来引用它。artifactId和groupId的组合必须是唯一的。换句话说，你不能有两个不同的项目拥有同样的artifactId和groupId；在某个特定的groupId下，artifactId也必须是唯一的。

> **注意**
>
> 虽然‘.’在groupId中很常用，而你应该避免在artifactId中使用它。因为在解析一个完整限定名字至子模块的时候，这会引发问题。

version
: 当一个构件发布的时候，它是使用一个版本号发布的。该版本号是一个数字定义符如“1.0”，“1.1.1”，或“1.1.2-alpha-01”。你也可以使用所谓的快照

[1] http://repo1.maven.org/maven2/org/apache/maven

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

（snapshot）版本。一个快照版是一个处于开发过程中的组件的版本，快照版本号通常以SNAPSHOT结尾；如，“1.0-SNAPSHOT”，“1.1.1-SNAPSHOT”，和“1-SNAPSHOT”。第 9.3.1 节 “项目版本”介绍了版本和版本界限。

还有第四个，也是最少用到的限定符：

classifier
: 如果你要发布同样的代码，但是由于技术原因需要生成两个单独的构件，你就要使用一个分类器（classifier）。例如，如果你想要构建两个单独的构件成JAR，一个使用Java 1.4编译器，另一个使用Java 6编译器，你就可以使用分类器来生成两个单独的JAR构件，它们有同样的groupId:artifactId:version组合。如果你的项目使用本地扩展类库，你可以使用分类器为每一个目标平台生成一个构件。分类器常用于打包构件的源码，JavaDoc或者二进制集合。

当我们在本书说到依赖的时候，我们通常使用如下的简短标志来描述一个依赖：groupId:artifactId:version。要引用Spring Framework的2.5版本，我们可以使用org.springframework:spring:2.5。当你要求Maven使用Maven Dependency插件打印出依赖列表的时候，你也会看到Maven倾向于使用这种简短的依赖标志来打印日志信息。

9.5.2. 多模块项目

多模块项目是那些包含一系列待构建模块的项目。一个多模块项目的打包类型总是pom，很少生成一个构件。一个模块项目的存在只是为了将很多项目归类在一起，成为一个构建。图 9.3 “多模块项目关系”展示了一个项目层次，它包含了两个打包类型为pom的父项目，另外三个项目的打包类型是jar：

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

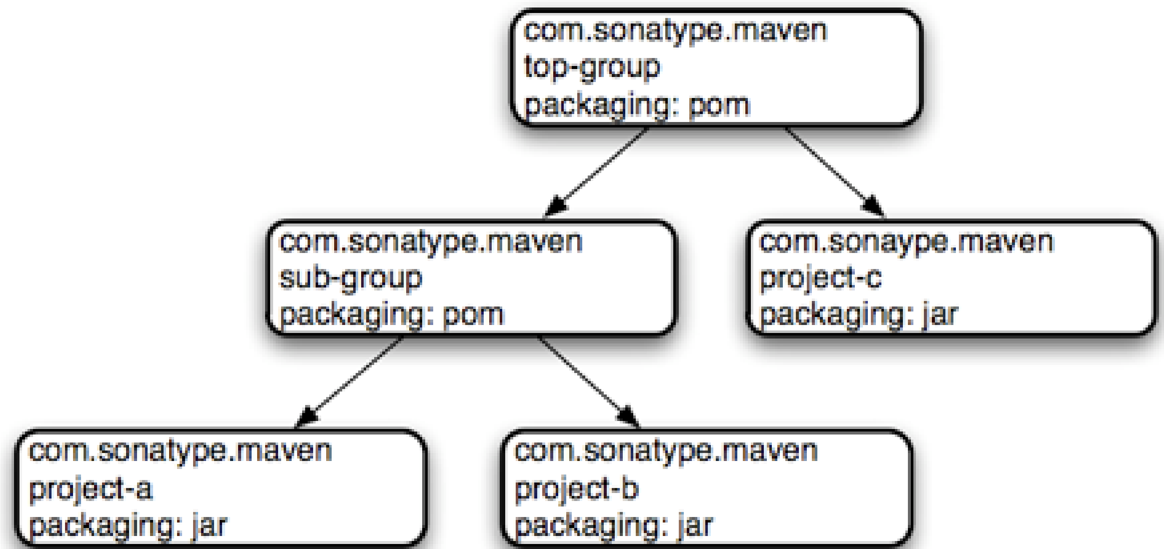


图 9.3. 多模块项目关系

文件系统上的目录结构也反映了该模块关系。图 9.3 “多模块项目关系”中的一组项目拥有如下的目录结构：

```
top-group/pom.xml
top-group/sub-group/pom.xml
top-group/sub-group/project-a/pom.xml
top-group/sub-group/project-b/pom.xml
top-group/project-c/pom.xml
```

这些项目相互关联，因为在POM中top-group和sub-group引用了子模块。例如，项目org.sonatype.mavenbook:top-group是一个打包类型为pom的多模块项目。该项目的pom.xml包含如下的modules元素：

例 9.10. top-group的modules元素

```
<project>
  <groupId>org.sonatype.mavenbook</groupId>
  <artifactId>top-group</artifactId>
  ...
  <modules>
    <module>sub-group</module>
    <module>project-c</module>
  </modules>
  ...
</project>
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

当Maven读取top-group的POM的时候，它会检查它的modules元素，看到top-group引用了项目sub-group和project-c。之后Maven会在它们的每个子目录中寻找pom.xml。Maven为每一个子模块重复这个过程：它会读取sub-group/pom.xml然后看到sub-group项目通过如下的modules元素引用了两个项目。

例 9.11. sub-group的modules元素

```
<project>
  ...
  <modules>
    <module>project-a</module>
    <module>project-b</module>
  </modules>
  ...
</project>
```

注意我们称多模块项目下的项目为“模块”而不是“子项目”。这是有目的的，是为了而不将由多模块项目归类的项目与那些从其它项目继承POM信息的项目混淆。

9.5.3. 项目继承

有些情况你会想要一个项目从父POM中继承一些值。你可能正构建一个大型的系统，你不想一遍又一遍的重复同样的依赖元素。如果你的项目通过parent元素使用继承，你就可以避免这种重复。当一个项目声明一个parent的时候，它从父项目的POM中继承信息。它也可以覆盖父POM中的值，或者添加一些新的值。

所有的Maven POM从父POM中继承值。如果一个POM没有通过parent元素指定一个直接的父项目，那这个POM就会从超级POM继承值。例 9.12 “项目继承”展示了a-parent的parent元素，它继承了a-parent项目定义的POM。

例 9.12. 项目继承

```
<project>
  <parent>
    <groupId>com.training.killerapp</groupId>
    <artifactId>a-parent</artifactId>
    <version>1.0-SNAPSHOT</version>
  </parent>
  <artifactId>project-a</artifactId>
  ...
</project>
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

在project-a中运行会mvn help:effective-pom显示一个POM，该POM合并了超级POM，a-parent中定义的POM，以及project-a中定义的POM。project-a显式的和隐式的继承关系如图 9.4 “a-parent和project的项目继承关系”所示：

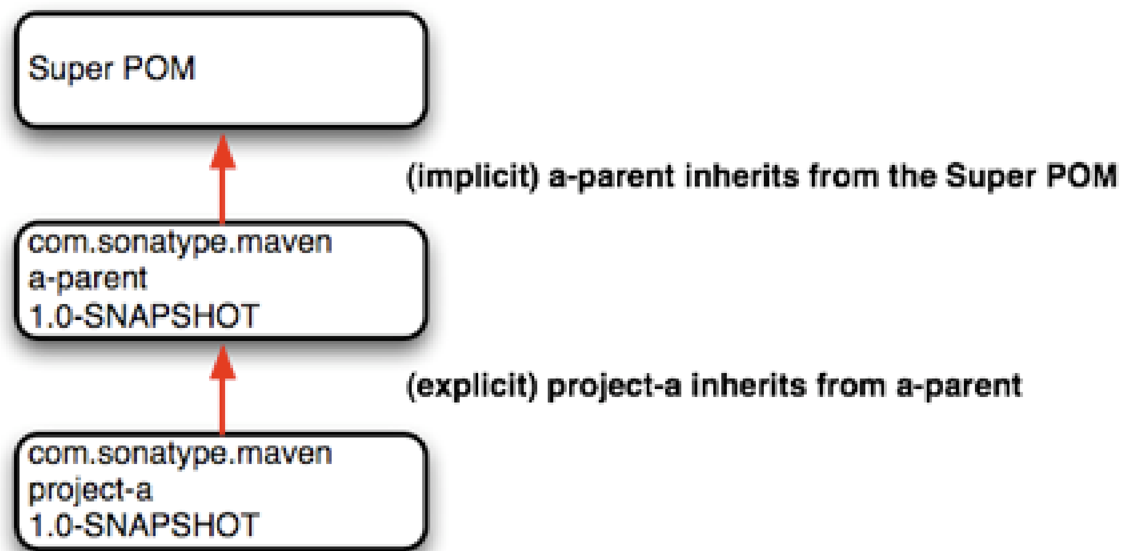


图 9.4. a-parent和project的项目继承关系

当一个项目指定一个父项目的时候，Maven在读取当前项目的POM之前，会使用这个父POM作为起始点。它继承所有东西，包括groupId和version。你会注意到project-a没有指定groupId和version，它们从a-parent继承而来。有了parent元素，一个POM就只需要定义一个artifactId。但这不是强制的，project-a可以有一个不同的groupId和version，但如果不提供值，Maven就会使用在父POM中指定的值。如果你开始使用Maven来管理和构建大型的多模块项目，你就会常常创建许多共享一组通用的groupId和version的项目。

当你继承一个POM，你可以选择直接使用继承的POM信息，或者选择覆盖它。以下是一个Maven POM从它父POM中继承的项目列表：

- 定义符（groupId和artifactId中至少有一个必须被覆盖）
- 依赖
- 开发者和贡献者
- 插件列表

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

- 报告列表
- 插件执行 （id匹配的执行会被合并）
- 插件配置

当Maven继承依赖的时候，它会将父项目中定义的依赖添加到子项目中。你可以使用Maven的这一特征来指定一些在所有项目被广泛使用的依赖，让它们从顶层POM中继承。例如，如果你的系统全局使用Log4J日志框架，你可以在你的顶层POM中列出该依赖。任何从该项目继承POM信息的项目会自动拥有Log4J依赖。类似的，如果你能确定每个项目都在使用同样版本的一个Maven插件，你可以在顶层父POM的pluginManagement元素中显式的列出该Maven插件的版本。

Maven假设父POM在本地仓库中可用，或者在当前项目的父目录(../pom.xml) 中可用。如果两个位置都不可用，默认行为还可以通过relativePath元素被覆盖。例如，一些组织更喜欢一个平坦的项目结构，父项目的pom.xml并不在子项目的父目录中。它可能在项目的兄弟目录中。如果你的子项目在目录./project-a中，父项目在名为./a-parent的目录中，你可以使用如下的配置来指定parent-a的POM的相对位置。

```
<project>
  <parent>
    <groupId>org.sonatype.mavenbook</groupId>
    <artifactId>a-parent</artifactId>
    <version>1.0-SNAPSHOT</version>
    <relativePath>../a-parent/pom.xml</relativePath>
  </parent>
  <artifactId>project-a</artifactId>
</project>
```

9.6. POM最佳实践

Maven可以用来管理那些简单的单模块系统，或者复杂到拥有数百个相互关联的子模块的项目。学习Maven过程的有一部分不仅仅是弄清楚Maven配置的语法，而是学习“Maven方式”——一组使用Maven组织和构建项目的最佳实践。本节试图展现一些这样的知识来帮助你采用最佳实践，你就不用从头开始去Maven的邮件列表的数年的内容中寻找这些技巧。

9.6.1. 依赖归类

如果你有一组逻辑上归类在一起的依赖。你可以创建一个打包方式为pom项目来将这些依赖归在一起。例如，让我们假设你的应用程序使用Hibernate，一种流行的对象关系映射框架。所有使用Hibernate的项目可能同时依赖于Spring Framework和MySQL JDBC驱动。你可以创建一个特殊的POM，它除了声明一组通用依赖之外什么也不做。这样你就不需要在每个使用Hibernate，Spring和MySQL的项目中包含所有这些依赖。你可以创

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

建一个项目叫做persistence-deps（持久化依赖的简称），然后让每个需要持久化的项目依赖于这个提供便利的项目。

例 9.13. 在一个单独的POM项目中巩固依赖

```
<project>
  <groupId>org.sonatype.mavenbook</groupId>
  <artifactId>persistence-deps</artifactId>
  <version>1.0</version>
  <packaging>pom</packaging>
  <dependencies>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate</artifactId>
      <version>${hibernateVersion}</version>
    </dependency>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-annotations</artifactId>
      <version>${hibernateAnnotationsVersion}</version>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-hibernate3</artifactId>
      <version>${springVersion}</version>
    </dependency>
    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <version>${mysqlVersion}</version>
    </dependency>
  </dependencies>
  <properties>
    <mysqlVersion>(5.1,)</mysqlVersion>
    <springVersion>(2.0.6,)</springVersion>
    <hibernateVersion>3.2.5.ga</hibernateVersion>
    <hibernateAnnotationsVersion>3.3.0.ga</hibernateAnnotationsVersion>
  </properties>
</project>
```

如果你在一个名为persistence-deps的目录下创建了这个项目，你需要做的只是创建该pom.xml并且运行mvn install。由于打包类型是pom，这个POM被安装到你的本地仓库。你现在就可以添加这个项目作为一个依赖，所有该项目的依赖就会被添加到你的项目中。当我们声明一个对于persistence-deps项目的依赖的时候，不要忘了指定依赖类型为pom。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

例 9.14. 声明一个对于POM的依赖

```
<project>
  <description>This is a project requiring JDBC</description>
  ...
  <dependencies>
    ...
    <dependency>
      <groupId>org.sonatype.mavenbook</groupId>
      <artifactId>persistence-deps</artifactId>
      <version>1.0</version>
      <type>pom</type>
    </dependency>
  </dependencies>
</project>
```

如果之后你决定切换一个不同的JDBC驱动（比如，JTDS），只要替换persistence-deps项目中的依赖，使用sourceforge.jtds:jtds而不再是mysql:mysql-java-connector，然后更新版本号。所有依赖于persistence-deps的项目，如果它们决定更新一个新的依赖版本，就会使用JTDS。巩固相互关联的依赖是一种减少pom.xml文件长度的很好的方法。如果你需要在项目间共享一组很多的依赖，你也可以建立在项目间建立父子关系，然后将所有共同的依赖重构到父项目中，但是这种父子方式的缺点是一个项目只能有一个父项目。有时候将类似的依赖归类在一起并且使用pom依赖是更明智的做法。因为这样你的项目就可以根据需要引用很多巩固依赖POM。

注意

当Maven使用一种“最近者胜出”方式解决依赖的时候，它会用到依赖的深度。当使用上述提到的依赖归类技术的时候，会把依赖推入整个树的更深一层。当在选择用pom归类依赖或者用父POM的dependenctManagement的时候，需要留意这一点。

9.6.2. 多模块 vs. 继承

继承于一个父项目和被一个多模块项目管理是有区别的。一个父项目是指它把所有的值传给它的子项目。一个多模块项目只是说它管理一组子模块，或者说一组子项目。多模块关系从上层往下定义。当建立一个多模块项目的时候，你告诉一个项目它的构建需要包含指定的模块。多模块构建用来将模块聚集到一个单独的构建中。父子关系是从叶节点往上定义的。父子关系更多的是处理一个特定项目的定义。当你给一个子项目关联一个父项目的时候，你告诉Maven该项目的POM起源于另一个项目。

为了展示选择继承还是多模块的决策过程，考虑如下的两个例子：用来生成本书的Maven项目，以及一个包含很多逻辑上同组模块的假想项目。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

9.6.2.1. 简单项目

首先，我们看一下maven-book项目。它的继承和多模块关系如图 9.5 “maven-book 多模块 vs. 继承”所示。

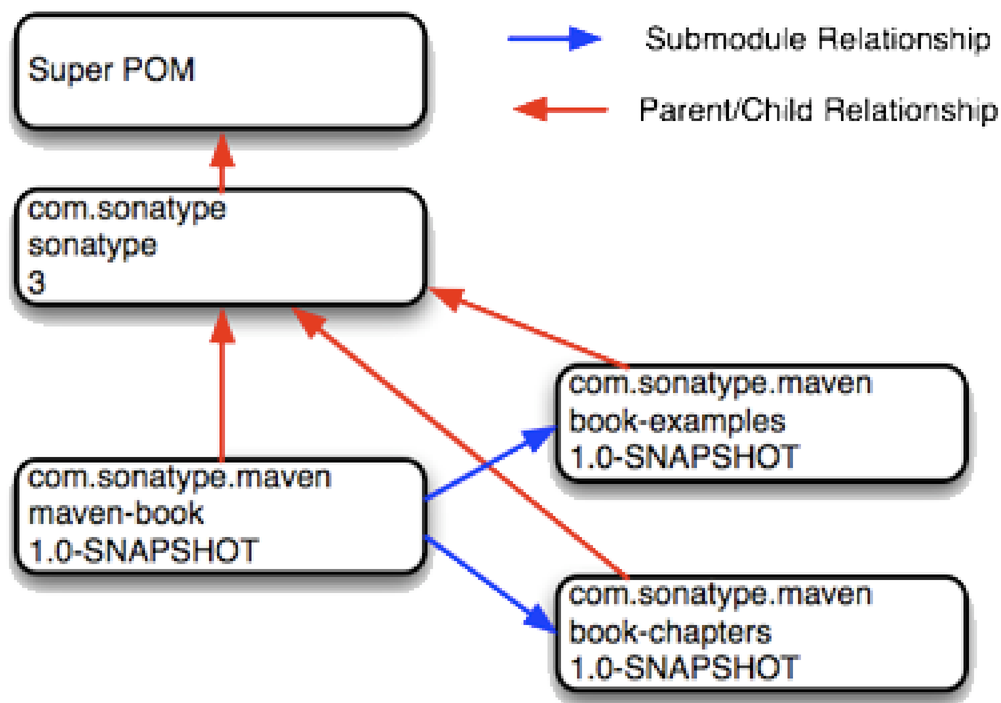


图 9.5. maven-book 多模块 vs. 继承

当我们构建你正阅读的Maven书的时候，我们在名为maven-book的项目下运行mvn package。该多模块项目包含两个子模块：book-examples和book-chapters。两者都共享同样的父项目，它们只通过同为maven-book子项目的方式关联。book-examples构建了可下载的本书样例的ZIP和TGZ存档文件。当我们在book-examples/目录使用mvn package运行book-examples的构建的时候，它完全不知道它是maven-book项目的一部分。book-examples完全不关心maven-book，它知道的是它的父项目是最顶层的sonatype POM，它自身创建样例的归档文件。该例中，maven-book的存在只是为了方便聚集模块构建。

该些书的项目没有定义一个父项目。所有这个三个项目：maven-book，book-examples，book-chapters和都继承同一个共享的“团体”父项目——sonatype。这是一种采用Maven的组织中常见的实践，一些组织定义一个顶层的团体POM，作为一个默认的父项目为其它项目服务，而不是让每个项目默认去扩展超级POM。在这个书本样例中，并不是一定要让book-examples和book-chapters共享同样的父POM，它们是完全不同的两个项目，拥有完全不同的而依赖，不同的构建配置，使用极为不同的插件创建你正阅读

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

的内容。“团体”POM能让组织有机会自定义一些Maven的默认行为，提供一些组织特定的信息，如配置部署设置和构建profile。

9.6.2.2. 多模块企业级项目

让我们看一下另一个例子，它提供了现实项目中继承和多模块关系存在的更准确的画面。图 9.6 “企业级多模块 vs. 继承”展示了类似于典型企业应用中的一组项目。有一个的公司顶层POM，其`artifactId`值为`sonatype`。有一个名为`big-system`的多模块项目，引用了子模块`server-side`和`client-side`。

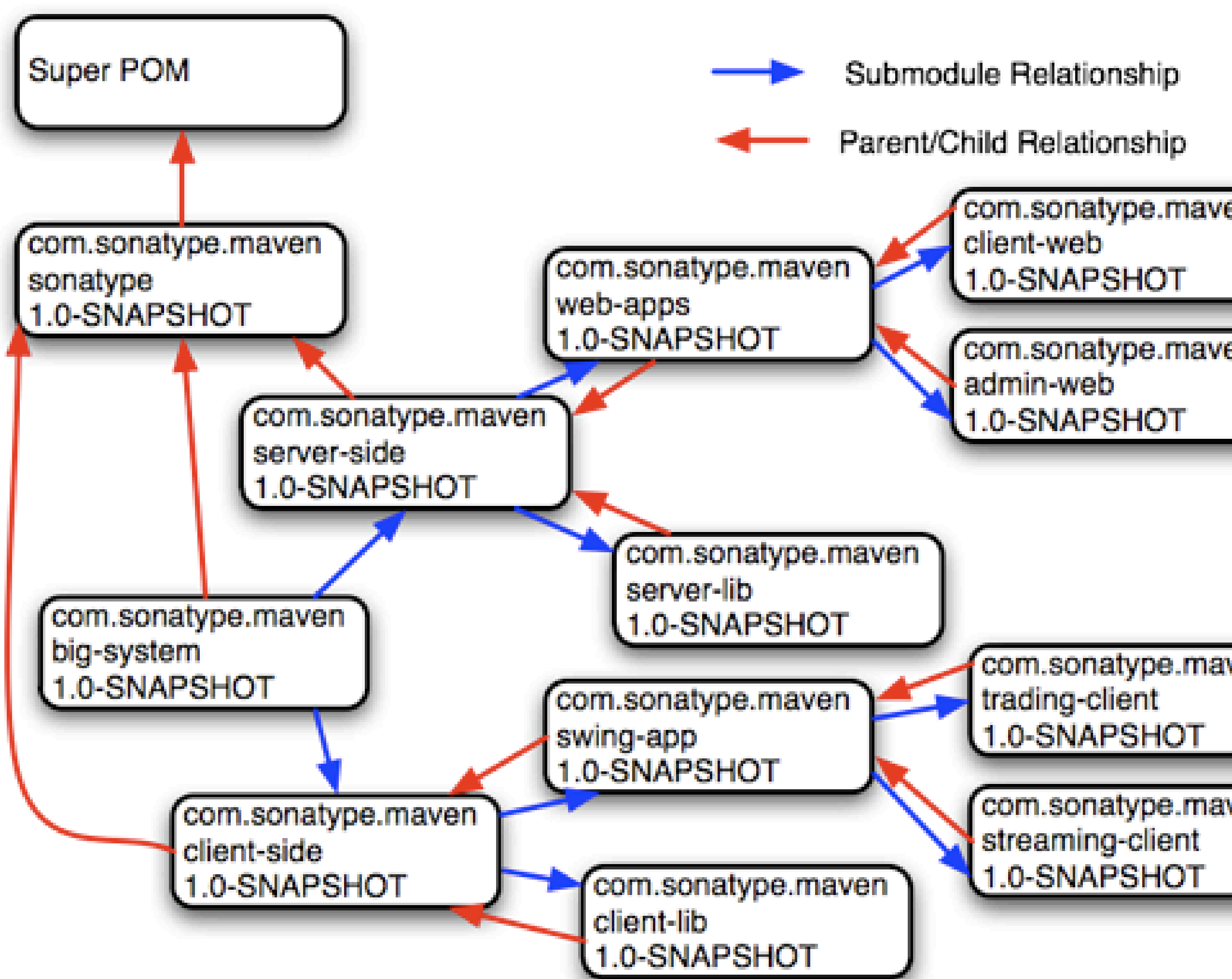


图 9.6. 企业级多模块 vs. 继承

这里到底是怎么回事呢？让我们尝试着给这一组混乱的箭头解构。首先，看一下`big-system`。这个`big-system`可能就是你将要运行mvn package以构建并测试整个系统的地

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

方。big-system引用了子模块client-side和server-side。这两个项目都管理了大量运行在服务端或者客户端的代码。让我们看一下server-side项目。在server-side下面有一个名为server-lib的项目和一个名为web-apps的多模块项目。在web-apps下面有两个Java web应用：client-web和admin-web。

让我们从client-web和admin-web到web-apps开始讨论父子关系。由于这两个web应用都用同样的web应用框架实现（假设是Wicket），两个项目都共享同样的一组核心依赖。对Servlet API，JSP API，和Wicket的依赖可以放到web-apps项目中。client-web和admin-web都需要依赖server-lib，它就可以定义为一个web-apps和server-lib之间的依赖。因为client-web和admin-web通过继承web-apps共享了如此多的配置，它们的POM很小，只包含定义符，父项目声明和最终构建名称。

本例中，使用父子关系的最主要原因是为了给一组逻辑关联的项目共享依赖和通用配置。所有big-system下的项目都通过子模块与其它项目关联，但是并不是所有子模块都被配置成指回该父项目。所有模块都是子模块是为了方便，要构建整个系统，只要到big-system项目目录下运行mvn package。再仔细看下上图，你会发现在server-side和big-system之间没有父子关联。这是为什么？POM继承十分有用，但它可能被滥用。当然在需要共享依赖和配置的时候，父子关联需要被使用。但当两个项目截然不同的时候使用父子关联是不明智的。举个例子，server-side和client-side项目。在系统中，让server-side和client-side都从big-system继承通用的POM是可能的，但一旦这两个子项目的重大分歧出现，你就需要费脑子一方面将构建配置抽离到big-system中，另一方面又需要不影响其它的子项目。即使server-side和client-side同时依赖于Log4J，它们也可能拥有截然不同的插件配置。

有时候基于风格和经验，为了允许项目如server-side和client-side保持完全独立，少量的重复配置是最小的代价。设计一组继承了五六层POM的第三方项目永远都不是一个好主意。这样的配置下，你可能不再需要在多个地方重复Log4J依赖，但你会需要查看五六个POM来弄清Maven如何计算出你的有效POM。所有的这些的新的复杂度只是为了避免五行依赖声明。在Maven中，有一种“Maven方式”，但是也有很多其它方式完成同样的事情。这都可归结为一种偏好和风格。大部分情况下，如果你的子模块定义了往回的父项目引用，不会出什么问题，但是你的Maven使用情况一直在变。

9.6.2.3. 原型父项目

如图 9.7 “为特定的项目使用父项目作为“原型””的例子所示，这是又一种假想的创造性的方式，使用继承和多模块构建达到重用依赖的目的。

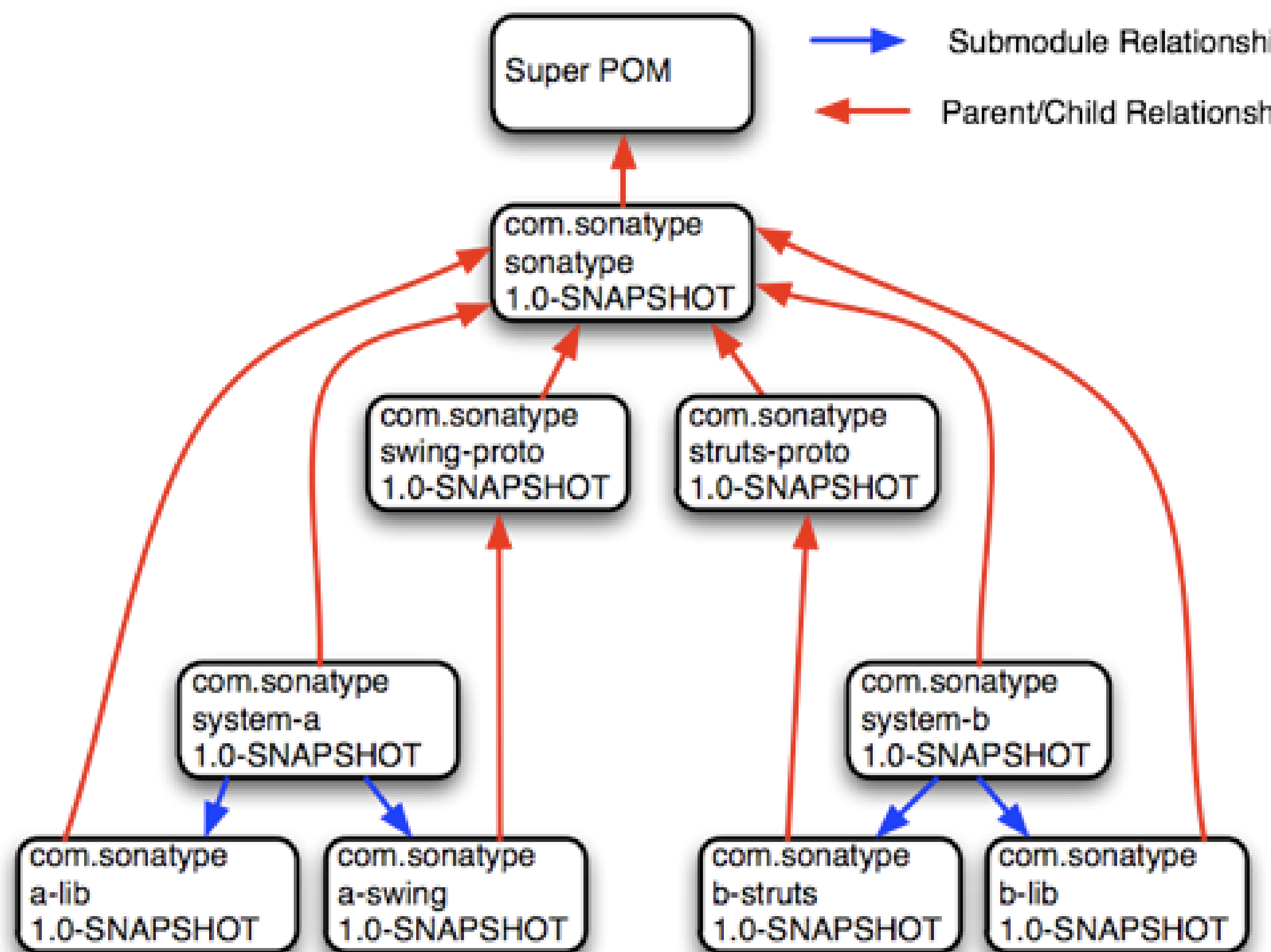


图 9.7. 为特定的项目使用父项目作为“原型”

在该例中，你有两个截然不同的系统：system-a和system-b，各自定义独立的应用。system-a定义了两个模块a-lib和a-swing。system-a和a-lib两者都定义了顶层的sonatype POM作为父项目，但a-swing项目定义了swing-proto作为它的父项目。在该系统中，swing-proto为所有Swing应用程序提供了一个基础POM，而struts-proto为所有Struts 2 web应用程序提供了一个基础POM。sonatype POM提供了高层的信息如groupId，组织信息和构建profile，struts-proto定义了所有创建struts应用需要的依赖。如果你的开发根据不同的应用有不同的特征，每类应用又需要遵循同样一组规则，那么这里介绍的方法很有用。如果你正创建很多struts应用，但是它们很少相互关联，你可能只需要在struts-proto中定义所有通用的东西。这种方式的缺点是有不能在system-a和system-b项目层次中使用父子关系来共享如开发人员和其它构建配置信息。一个项目只能有一个父项目。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

这种方法的另外一个缺点是，一旦你有一个项目需要“破坏该模型”，你需要重写父POM，或者想办法将自定义信息提取到一个共享的父项目中，而不让这些自定义信息影响所有子项目。总得来说，为特定的项目“类型”使用POM作为原型不是一种推荐的做法。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

第 10 章 构建生命周期

10.1. 简介

Maven使用POM描述项目，将其建模成一些名词。POM记录了一个项目的定义：项目包含什么？需要怎样的打包类型？是否包含一个父项目？它的依赖是什么？我们已经在前面的章节展示了如何描述一个项目，但还没有介绍一种允许Maven针对这些对象进行操作的机制。在Maven中这些“动词”是由Maven插件包装的一些目标，它们绑定到一个构建生命周期的阶段中。一个Maven生命周期包含了一些有序的命名阶段：prepare-resources，compile，package，和install以及其它。有个阶段抽象了编译过程，有个阶段抽象了打包过程。而那些pre-和post-阶段可以用来注册一些必须在某些特定阶段之前或之后运行的目标。当你让Maven构建一个项目的时候，你其实是让它一步步通过那些预定义的有序的阶段，并且运行所有注册到某个特定阶段的目标。

一个构建生命周期是一组精心组织的有序的阶段，它的存在能使所有注册的目标变得有序运行。这些目标根据项目的打包类型被选择并绑定。Maven中有三种标准的生命周期：清理（clean），默认（default）(有时候也称为构建)，和站点（site）。本章，我们将学习Maven如何将目标绑定到生命周期，生命周期如何自定义。你同时也会学到默认生命周期阶段的知识。

10.1.1. 清理生命周期 (clean)

第一个你将感兴趣的生命周期是Maven中最简单的生命周期。运行mvn clean将调用清理生命周期，它包含了三个生命周期阶段：

- pre-clean
- clean
- post-clean

在清理生命周期中最有意思的阶段是clean阶段。Clean插件的clean目标（clean:clean）被绑定到清理生命周期中的clean阶段。目标clean:clean通过删除构建目录删除整个构建的输出。如果你没有自定义构建目录位置，那么构建目录就是定义在超级POM中的/usr/local/hudson/hudson-home/jobs/maven-guide-zh-to-production/workspace/content-zh/target。当你运行clean:clean目标的时候你并不是直接运行mvn clean:clean，你可以通过执行清理生命周期的clean阶段运行该目标。运行clean阶段能让Maven有机会执行其它可能被绑定到pre-clean阶段的目标。

例如，假设你想要在pre-clean的时候触发一个antrun:run目标任务来输出一个通知，或者需要在项目构建目录被删除之前将其归档。简单的运行clean:clean目标不会完整的执行该生命周期，但是指定clean阶段就能使用clean生命周期，并且逐个的经过生命周

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

期阶段，直到到达clean阶段。例 10.1 “在pre-clean阶段触发一个目标”展示了一个样例，在它的构建配置中，绑定了antrun:run至pre-clean阶段，输出一个警告告诉用户项目构件即将被删除。该例中，antrun:run目标被用来执行一些随意的Ant命令来检查项目的构件。如果项目的构件将要被删除，它会打印该信息至屏幕。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

```
<project>
  ...
  <build>
    <plugins>... <plugin>
    <artifactId>maven-antrun-plugin</artifactId>
    <executions>
      <execution>
        <id>file-exists</id>
        <phase>pre-clean</phase>
        <goals>
          <goal>run</goal>
        </goals>
        <configuration>
          <tasks>
            <!-- adds the ant-contrib tasks (if/then/else used below) -->
            <taskdef resource="net/sf/antcontrib/antcontrib.properties" />
            <available
              file="/usr/local/hudson/hudson-home/jobs/maven-guide-zh-to-production
              property="file.exists" value="true" />

            <if>
              <not>
                <isset property="file.exists" />
              </not>
              <then>
                <echo>No
                  book.jar to
                  delete</echo>
              </then>
              <else>
                <echo>Deleting
                  book.jar</echo>
              </else>
            </if>
          </tasks>
        </configuration>
      </execution>
    </executions>
    <dependencies>
      <dependency>
        <groupId>ant-contrib</groupId>
        <artifactId>ant-contrib</artifactId>
        <version>1.0b2</version>
      </dependency>
    </dependencies>
  </plugin>
  </plugins>
  </build>
</project>
```

在带有如上构建配置的项目中运行mvn clean会生成如下的输出：

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛


```
[INFO] Scanning for projects...
[INFO] ----------------------------------------------------------------------
[INFO] Building Your Project
[INFO]    task-segment: [clean]
[INFO] ----------------------------------------------------------------------
[INFO] [antrun:run {execution: file-exists}]
[INFO] Executing tasks
     [echo] Deleting your-project-1.0-SNAPSHOT.jar
[INFO] Executed tasks
[INFO] [clean:clean]
[INFO] Deleting directory ~/corp/your-project/target
[INFO] Deleting directory ~/corp/your-project/target/classes
[INFO] Deleting directory ~/corp/your-project/target/test-classes
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 1 second
[INFO] Finished at: Wed Nov 08 11:46:26 CST 2006
[INFO] Final Memory: 2M/5M
[INFO] ------------------------------------------------------------------------
```

除了在pre-clean阶段配置Maven去运行一个目标，你也可以自定义Clean插件去删除构建输出目录以外的文件。你可以配置该插件去删除那些在fileSet中指定的文件。下面的样例配置了Clean插件，使用标准的Ant文件通配符：*和**，删除所有target-other/目录中的.class文件。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

例 10.2. 自定义Clean插件的行为

```
<project>
  <modelVersion>4.0.0</modelVersion>
  ...
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-clean-plugin</artifactId>
        <configuration>
          <filesets>
            <fileset>
              <directory>target-other</directory>
              <includes>
                <include>*.class</include>
              </includes>
            </fileset>
          </filesets>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

10.1.2. 默认生命周期 (default)

大部分Maven用户将会对默认生命周期十分熟悉。它是一个软件应用程序构建过程的总体模型。第一个阶段是validate，最后一个阶段是deploy。这些默认Maven生命周期的阶段如表 10.1 “Maven默认生命周期阶段”所示：

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

表 10.1. Maven默认生命周期阶段

生命周期阶段	描述
validate	验证项目是否正确，以及所有为了完整构建必要的信息是否可用
generate-sources	生成所有需要包含在编译过程中的源代码
process-sources	处理源代码，比如过滤一些值
generate-resources	生成所有需要包含在打包过程中的资源文件
process-resources	复制并处理资源文件至目标目录，准备打包
compile	编译项目的源代码
process-classes	后处理编译生成的文件，例如对Java类进行字节码增强（bytecode enhancement）
generate-test-sources	生成所有包含在测试编译过程中的测试源码
process-test-sources	处理测试源码，比如过滤一些值
generate-test-resources	生成测试需要的资源文件
process-test-resources	复制并处理测试资源文件至测试目标目录
test-compile	编译测试源码至测试目标目录
test	使用合适的单元测试框架运行测试。这些测试应该不需要代码被打包或发布
prepare-package	在真正的打包之前，执行一些准备打包必要的操作。这通常会产生一个包的展开的处理过的版本（将会在Maven 2.1+中实现）
package	将编译好的代码打包成可分发的格式，如JAR，WAR，或者EAR
pre-integration-test	执行一些在集成测试运行之前需要的动作。如建立集成测试需要的环境
integration-test	如果有必要的话，处理包并发布至集成测试可以运行的环境
post-integration-test	执行一些在集成测试运行之后需要的动作。如清理集成测试环境。
verify	执行所有检查，验证包是有效的，符合质量规范
install	安装包至本地仓库，以备本地的其它项目作为依赖使用
deploy	复制最终的包至远程仓库，共享给其它开发人员和项目（通常和一次正式的发布相关）

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

10.1.3. 站点生命周期 (site)

Maven不仅仅能从一个项目构建软件构件，它还能为一个或者一组项目生成项目文档和报告。项目文档和站点生成有一个专有的生命周期，它包含了四个阶段：

1. pre-site
2. site
3. post-site
4. site-deploy

默认绑定到站点生命周期的目标是：

1. site - site:site
2. site-deploy -site:deploy

打包类型通常不更改此生命周期，因为打包类型主要和构件创建有关，和站点生成没有太大的关系。Site插件触发Doxia[1]执行文档生成，以及执行其它报告生成插件。你可以通过运行如下命令从一个Maven项目生成一个站点：

```
$ mvn site
```

有关更多的Maven站点生成信息，查看第 15 章站点生成。

10.2. 打包相关生命周期

绑定到每个阶段的特定目标默认根据项目的打包类型设置。一个打包类型为jar的项目和一个打包类型为war的项目拥有不同的两组默认目标。packaging元素影响构建一个项目需要的步骤。举个打包如何影响构建的例子，考虑有两个项目：一个打包类型是pom，另外一个是jar。在package阶段，打包类型为pom的项目会运行site:attach-descriptor目标，而打包类型为jar的项目会运行jar:jar目标。

下面的小节描述了Maven中内建打包类型的生命周期。可以使用这些小节来找出哪些默认目标映射到了哪些默认生命周期阶段。

10.2.1. JAR

JAR是默认的打包类型，是最常用的，因此也就是生命周期配置中最经常遇到的打包类型。JAR生命周期默认的目标如表 10.2 “JAR打包默认的目标”所示：

[1] http://maven.apache.org/doxia/

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

表 10.2. JAR打包默认的目标

生命周期阶段	目标
process-resources	resources:resources
compile	compiler:compile
process-test-resources	resources:testResources
test-compile	compiler:testCompile
test	surefire:test
package	jar:jar
install	install:install
deploy	deploy:deploy

10.2.2. POM

POM是最简单的打包类型。不像一个JAR，SAR，或者EAR，它生成的构件只是它本身。没有代码需要测试或者编译，也没有资源需要处理。打包类型为POM的项目的默认目标如表 10.3 “POM打包默认的目标”所示：

表 10.3. POM打包默认的目标

生命周期阶段	目标
package	site:attach-descriptor
install	install:install
deploy	deploy:deploy

10.2.3. Maven Plugin

该打包类型和JAR打包类型类似，除了三个目标：plugin:descriptor，plugin:addPluginArtifactMetadata，和plugin:updateRegistry。这些目标生成一个描述文件，对仓库数据执行一些修改。打包类型为maven-plugin的项目的默认目标如表 10.4 “maven-plugin打包默认的目标”所示。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

表　10.4.　maven-plugin打包默认的目标

生命周期阶段	目标
generate-resources	plugin:descriptor
process-resources	resources:resources
compile	compiler:compile
process-test-resources	resources:testResources
test-compile	compiler:testCompile
test	surefire:test
package	jar:jar, plugin:addPluginArtifactMetadata
install	install:install, plugin:updateRegistry
deploy	deploy:deploy

10.2.4.　EJB

EJB，或者说企业Java Bean，是企业级Java中模型驱动开发的常见数据访问机制。Maven提供了对EJB 2和3的支持。你必须配置EJB插件来为EJB3指定打包类型，否则该插件默认认为EJB为2.1，并寻找某些EJB配置文件是否存在。打包类型为EJB的项目的默认目标如表　10.5 “EJB打包默认的目标”所示。

表　10.5.　EJB打包默认的目标

生命周期阶段	目标
process-resources	resources:resources
compile	compiler:compile
process-test-resources	resources:testResources
test-compile	compiler:testCompile
test	surefire:test
package	ejb:ejb
install	install:install
deploy	deploy:deploy

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

10.2.5. WAR

WAR打包类型和JAR以及EJB类似。例外是这里的package目标是war:war。注意war:war插件需要一个web.xml配置文件在项目的src/main/webapp/WEB-INF目录中。打包类型为WAR的项目的默认目标如表 10.6 “WAR打包默认的目标”所示。

表 10.6. WAR打包默认的目标

生命周期阶段	目标
process-resources	resources:resources
compile	compiler:compile
process-test-resources	resources:testResources
test-compile	compiler:testCompile
test	surefire:test
package	war:war
install	install:install
deploy	deploy:deploy

10.2.6. EAR

EAR可能是最简单的Java EE结构体，它主要包含一个部署描述符application.xml文件，一些资源和一些模块。EAR插件有个名为generate-application-xml的目标，它根据EAR项目POM的配置生成application.xml。打包类型为EAR的项目的默认目标如表 10.7 “EAR打包默认的目标” 所示。

表 10.7. EAR打包默认的目标

生命周期阶段	目标
generate-resources	ear:generate-application-xml
process-resources	resources:resources
package	ear:ear
install	install:install
deploy	deploy:deploy

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

10.2.7. 其它打包类型

以上列表并非是Maven中所有可用打包类型。有许多打包格式在外部的项目和插件中可用：NAR（本地归档）打包类型，用来生成Adobe Flash和Flex内容的项目的SWF和SWC打包类型，以及很多其它类型。你也可以自定义打包类型，定制默认的生命周期目标来适应你自己项目的打包需求。

为了使用自定义的打包类型，你需要两样东西：一个定义了定制打包类型生命周期的插件，和一个包含该插件的仓库。有些定制打包类型是由中央Maven仓库中可用的插件定义的。这里有一个样例项目，它引用了Israfil Flex插件，使用自定义打包类型SWF根据Adobe Flex生成输出。

例 10.3. 为Adobe Flex (SWF)定制打包类型

```
<project>
  ...
  <packaging>swf</packaging>
  ...
  <build>
    <plugins>
      <plugin>
        <groupId>net.israfil.mojo</groupId>
        <artifactId>maven-flex2-plugin</artifactId>
        <version>1.4-SNAPSHOT</version>
        <extensions>true</extensions>
        <configuration>
          <debug>true</debug>
          <flexHome>${flex.home}</flexHome>
          <useNetwork>true</useNetwork>
          <main>org/sonatype/mavenbook/Main.mxml</main>
        </configuration>
      </plugin>
    </plugins>
  </build>
  ...
</project>
```

在???中，我们向你展示了如何使用自定义的生命周期创建你自己的打包类型。该样例应该能让你了解到，引用一个定制的打包类型你需要做些什么。你需要做的只是引用那个提供定制打包类型的插件。Israfil Flex插件是托管在Google Code的第三方Maven插件，要了解更多的关于此插件的信息，以及如何使用Maven编译Adobe Flex，访问http://code.google.com/p/israfil-mojo。该插件为SWF打包类型提供了如下的生命周期。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

表 10.8. SWF打包的默认生命周期

生命周期阶段	目标
compile	flex2:compile-swc
install	install
deploy	deploy

10.3. 通用生命周期目标

很多打包类型的生命周期有类似的目标。如果你看一下绑定到WAR和JAR生命周期的目标，你会发现它们只有在package阶段有区别。WAR生命周期的package阶段调用了war:war，而JAR生命周期的package阶段调用了jar:jar。大部分你将要接触的生命周期共享一些通用生命周期目标，用来管理资源，运行测试，以及编译源代码。本节，我们会详细讨论这些通用的生命周期目标。

10.3.1. Process Resources

大部分生命周期将resources:resources目标绑定到process-resources阶段。process-resources阶段处理资源并将资源复制到输出目录。如果你没有自己自定义超级POM中的默认目录位置，Maven就会将/usr/local/hudson/hudson-home/jobs/maven-guide-zh-to-production/workspace/content-zh/src/main/resources中的文件复制到/usr/local/hudson/hudson-home/jobs/maven-guide-zh-to-production/workspace/content-zh/target/classes，或者是由/usr/local/hudson/hudson-home/jobs/maven-guide-zh-to-production/workspace/content-zh/target/classes定义的目录。除了复制资源文件至输出目录，Maven同时也会在资源上应用过滤器，能让你替换资源文件中的一些符号。就像在POM中我们通过MavenProject: org.sonatype.mavenbook:content-zh:0.6-SNAPSHOT @ /usr/local/hudson/hudson-home/jobs/maven-guide-zh-to-production/workspace/content-zh/pom.xml标记引用变量一样，你也可以使用同样的语法在你项目的资源文件中引用变量。与profile联系起来，这样的特性就能用来生成针对不同部署平台的构件。当我们需要为同一个项目的开发，测试，staging，以及产品平台环境生成输出的时候，该特性就十分有用。要了解更多的关于构建profile的信息，查看第 11 章构建Profile。

为了阐述资源过滤，假设你有个带有XML文件src/main/resources/META-INF/service.xml的项目。你想要提取出一些配置变量至一个属性文件。换句话说，你可能想要为你的数据库引用JDBC URL，用户名和密码，并且你不想将这些值直接放到service.xml文件里，而是想要使用一个属性文件来存储你程序中的所有配置点。这么做能让你将所有配置信息固定到单独的一个属性文件中，当你需要面对一个新的部署环境的时候，就很容易更改配置的值。首先，看一下src/main/resources/META-INF/service.xml的内容。

例 10.4. 在项目资源中使用属性

```
<service>
  <!-- This URL was set by project version 0.6-SNAPSHOT -->
  <url>${jdbc.url}</url>
  <user>${jdbc.username}</user>
  <password>${jdbc.password}</password>
</service>
```

该XML文件使用你在POM中用到的同样的属性引用语法，第一个引用的变量是project，它同时也是POM中的隐式变量。project变量提供了对POM信息的访问。接下来的三个变量引用是，jdbc.url，jdbc.username和jdbc.password。这些自定义的变量在一个属性文件src/main/filters/default.properties中定义。

例 10.5. src/main/filters中的default.properties

```
jdbc.url=jdbc:hsqldb:mem:mydb
jdbc.username=sa
jdbc.password=
```

要配置使用该default.properties文件的资源过滤，我们需要在这个项目的POM中指定两样东西：构建配置的filters元素中的属性文件列表，以及一个标记告诉Maven资源目录需要过滤。默认的Maven行为会跳过过滤，只是将资源复制到输出目录；你需要显式的配置资源过滤，否则Maven就会置之不理。这种Maven资源过滤的默认行为是为了确保不让Maven替换掉一些你不想替换的MavenProject: org.sonatype.mavenbook:content-zh:0.6-SNAPSHOT @ /usr/local/hudson/hudson-home/jobs/maven-guide-zh-to-production/workspace/content-zh/pom.xml引用。

例 10.6. 过滤资源（替换属性）

```
<build>
  <filters>
    <filter>src/main/filters/default.properties</filter>
  </filters>
  <resources>
    <resource>
      <directory>src/main/resources</directory>
      <filtering>true</filtering>
    </resource>
  </resources>
</build>
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

正如Maven中所有目录一样，资源目录并非一定要在src/main/resources。这只是定义在超级POM中的默认值。你应该也注意到你不需要将你所有的资源合并到一个单独的目录中。你可以将资源分离至src/main目录下的独立的目录中。假设你有个项目包含了数百个XML文档和数百个图片。你可能希望创建两个目录src/main/xml和src/main/images来存储这些内容，而不是将它们混合在src/main/resources目录中。为了添加资源目录列表，你需要在你的构建配置中加入如下的resource元素：

例　10.7.　配置额外的资源目录

```
<build>
  ...
  <resources>
    <resource>
      <directory>src/main/resources</directory>
    </resource>
    <resource>
      <directory>src/main/xml</directory>
    </resource>
    <resource>
      <directory>src/main/images</directory>
    </resource>
  </resources>
  ...
</build>
```

当你构建一个项目用来生成控制台程序或者命令行工具的时候，你通常发现自己正编写一个shell脚本，需要引用构建生成的JAR。当你使用assembly插件为一个应用程序生成如ZIP或TAR的分发包的时候，你可能会将所有的脚本放到如src/main/command的目录下。在下面的POM资源配置中，你会看到我们如何使用资源过滤器和一个对项目变量的引用，生成JAR的最终名称。要了解更多的关于Maven Assembly插件的信息，参考第　12　章Maven套件。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

例 10.8. 过滤脚本资源

```
<build>
  <groupId>org.sonatype.mavenbook</groupId>
  <artifactId>simple-cmd</artifactId>
  <version>2.3.1</version>
  ...
  <resources>
    <resource>
      <filtering>true</filtering>
      <directory>/usr/local/hudson/hudson-home/jobs/maven-guide-zh-to-production/wo
      <includes>
        <include>run.bat</include>
        <include>run.sh</include>
      </includes>
      <targetPath>/usr/local/hudson/hudson-home/jobs/maven-guide-zh-to-production/w
    </resource>
    <resource>
      <directory>/usr/local/hudson/hudson-home/jobs/maven-guide-zh-to-production/wo
    </resource>
  </resources>
  ...
</build>
```

如果你在该项目中运行mvn process-resources，最后你会在/usr/local/hudson/hudson-home/jobs/maven-guide-zh-to-production/workspace/content-zh中得到两个文件，run.sh和run.bat，我们在resource元素中挑选出这两个文件，配置过滤器，然后设置targetPath为/usr/local/hudson/hudson-home/jobs/maven-guide-zh-to-production/workspace/content-zh。在第二个resource元素中，我们配置了默认资源路径，使其在不做过滤的情况下被复制到默认输出目录。例 10.8 "过滤脚本资源"告诉你如何声明两个资源目录，如何给它们提供不同的过滤配置和目标目录配置。例 10.8 "过滤脚本资源"项目的src/main/command目录下包含了一个含有如下内容的run.bat文件。

```
@echo off
java -jar book.jar %*
```

在运行了mvn process-resources之后，目录/usr/local/hudson/hudson-home/jobs/maven-guide-zh-to-production/workspace/content-zh下会有一个含有如下内容的名为run.bat的文件：

```
@echo off
java -jar simple-cmd-2.3.1.jar %*
```

在带有很多不同种类资源的复杂的项目中，我们会发现将资源分离到多个目录下十分有利，原因之一就是我们可以为特定的资源子集自定义过滤器。针对不同的过滤需求，除

了将资源存储到不同的目录下，我们还可以使用更复杂的一组包含/排除模式来匹配那些符合模式的资源文件。

10.3.2. Compile

大部分生命周期将Compiler插件的compile目标绑定到compile阶段。该阶段会调用compile:compile，后者被配置成编译所有的源码并复制到构建输出目录。如果你没有自定义超级POM中的值，compile:compile将会编译src/main/java中的所有内容至target/classes。Compiler插件调用javac，使用的source设置为1.3，默认target设置为1.1。换句话说，Compiler插件会假设你所有的Java源代码遵循Java 1.3，目标为Java 1.1 JVM。如果你想要更改这些设置，你需要在POM中为Compiler插件提供source和target配置，如例 10.9 “为Compiler插件设置source和target版本”所示。

例 10.9. 为Compiler插件设置source和target版本

```
<project>
  ...
  <build>
    ...
    <plugins>
      <plugin>
        <artifactId>maven-compiler-plugin</artifactId>
        <configuration>
          <source>1.5</source>
          <target>1.5</target>
        </configuration>
      </plugin>
    </plugins>
    ...
  </build>
  ...
</project>
```

要注意我们配置的是Compiler插件，而不是compile:compile目标。如果我们只要为compile:compile目标配置source和target，就要将configuration元素放到compile:compile目标的execution元素下。我们为整个插件配置source和target，是因为compile:compile并不是我们唯一我们感兴趣配置的目标。当Maven使用compile:testCompile目标编译测试代码的时候，Compiler插件会被重用，因此在插件级别配置source和target，一次就能配置该插件所有的目标。

如果你想要自定义源码的位置，你也可以更改构建配置。如果你想要存储项目的源码至src/java而非src/main/java，让构建输出至classes而非target/classes，你可以覆盖定义在超级POM中的sourceDirectory的默认值。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

例 10.10. 覆盖默认的源码和输出目录

```
<build>
  ...
  <sourceDirectory>src/java</sourceDirectory>
  <outputDirectory>classes</outputDirectory>
  ...
</build>
```

警告

虽然让Maven屈服于你自己的项目目录结构可能看起来很有必要，但我们还是要不断强调你应该牺牲自己关于目录结构的想法，而遵循Maven的默认值。这不是说我们要给你洗脑，要你接受Maven的方式，这是因为如果你的项目遵循大部分的约定，别人会很容易理解你的项目。因此忘掉你自己的想法，别那样做。

10.3.3. Process Test Resources

process-test-resources阶段最难和process-resources阶段区别。在POM中它们有一些微小的差别，但大部分是一样的。你可以像过滤一般的资源那样过滤测试资源。测试资源的默认位置定义在超级POM中，为src/test/resources，默认的输出目录为target/test-classes，由/usr/local/hudson/hudson-home/jobs/maven-guide-zh-to-production/workspace/content-zh/target/test-classes定义。

10.3.4. Test Compile

test-compile阶段基本上和compile阶段一致。唯一的不同是会调用compile:testCompile编译测试源代码目录至测试构建构建输出目录。如果你没有在超级POM中自定义默认目录，compile:testCompile将会编译src/test/java中的源码至target/test-classes目录。

类似源代码目录，如果你想要自定义测试源码目录和测试编译输出目录的位置，你可以覆盖testSourceDirectory和testOutputDirectory。如果你想要将测试源代码存储在src-test/而非src/test/java，保存测试字节码至classes-test/而非target/test-classes，你可以使用如下的配置：

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

例 10.11. 覆盖测试源码和输出的位置

```
<build>
  ...
  <testSourceDirectory>src-test</testSourceDirectory>
  <testOutputDirectory>classes-test</testOutputDirectory>
  ...
</build>
```

10.3.5. Test

大部分生命周期绑定Surefire插件的test目标至test阶段。Surefire插件是Maven的单元测试插件，Surefire默认的行为是寻找测试源码目录下所有以*Test结尾的类，以JUnit[2]测试的形式运行它们。Surefire插件也可以配置成运行TestNG[3]单元测试。

运行过mvn test之后，你应该注意到Surefire在`target/surefire-reports`目录生成了许多报告。该目录内每个Surefire插件运行过的测试都会有相关的两个文件：一个是包含测试运行信息的XML文档，另一个是包含单元测试输出的文本文件。如果测试阶段有问题，单元测试失败了，你可以使用Maven的输出以及该目录下的内容来追查测试失败的原因。在站点生成的时候，`surefire-reports/`目录的内容会被用来创建报告，使项目所有单元测试的总体情况清晰明了。

如果你工作的项目有一些失败的单元测试，同时你想让项目生成输出，你需要配置Surefire插件在遇到失败的情况下继续一个构建。当遇到单元测试失败的时候，默认行为是停止构建。要覆盖这种行为，你需要设置Surefire插件的`testFailureIgnore`配置属性为true。

例 10.12. 配置Surefire忽略单元测试失败

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-surefire-plugin</artifactId>
     <configuration>
       <testFailureIgnore>true</testFailureIgnore>
     </configuration>
    </plugin>
    ...
  </plugins>
</build>
```

[2] http://www.junit.org

[3] http://www.testng.org

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

如果你想要整个的跳过测试，你可以运行如下的命令：

```
$ mvn install -Dmaven.test.skip=true
```

maven.test.skip变量同时控制Compiler和Surefire插件，如果你传入maven.test.skip，就等于告诉Maven整个的跳过测试。

10.3.6. Install

Install插件的install目标基本上都是绑定到install生命周期阶段。install:install目标只不过是将项目的主要构件安装到本地仓库。如果你有一个项目，groupId是org.sonatype.mavenbook，artifactId是simple-test，version是1.0.2，那么install:install目标就会从target/simple-test-1.0.2.jar复制JAR文件至~/.m2/repository/org/sonatype/mavenbook/simple-test/1.0.2/simple-test-1.0.2.jar。如果这个项目的打包类型是POM，那么该目标就仅仅复制POM到本地仓库。

10.3.7. Deploy

Deploy插件的deploy目标通常绑定到deploy生命周期阶段。该阶段用来将一个构件部署到远程Maven仓库，当你执行一次发布的时候通常需要更新远程仓库。一次部署过程可以简单到复制一个文件至另外一个目录，或者复杂到使用公钥通过SCP传送一个文件。部署设置通常包含远程仓库的证书，并且，这样的部署设置通常不会存储在pom.xml中。部署设置通常可以在用户单独的~/.m2/settings.xml中找到。到现在为止，你要知道的是deploy:deploy被绑定到deploy阶段，它会处理传送一个构件至发布仓库，更新一些可能被此次部署影响的仓库信息。

第 11 章 构建Profile

11.1. Profile是用来做什么的?

Profile能让你为一个特殊的环境自定义一个特殊的构建；profile使得不同环境间构建的可移植性成为可能。

不同的构建环境是什么意思？构建环境的两个例子是产品环境和开发环境。当你在开发环境中工作时，你的系统可能被配置成访问运行在你本机的开发数据库实例，而在产品环境中，你的系统被配置成从产品数据库读取数据。Maven能让你定义任意数量的构建环境（构建profile），这些定义可以覆盖pom.xml中的任何配置。你可以配置你的应用程序，在“开发”profile中，访问本地的开发数据库实例，在“产品”profile中，访问产品数据库。Profile也可以通过环境和平台被激活，你可以自定义一个构建，它根据不同的操作系统或者不同的JDK版本有不同的行为。在我们讨论使用和配置Maven profile之前，我们需要定义构建可移植性的概念。

11.1.1. 什么是构建可移植性

一个构建的“可移植性”是指将一个项目在不同的环境中构建的难易度。一个不用做任何自定义配置或者属性文件配置就能工作的构建，比一个需要很多配置才能工作的构建，具有更高的可移植性。构建可移植性最高的项目往往是一些开源项目如Apache Commons或者Apache Velocity，它们自带Maven构建配置，不需要或者需要很少的自定义配置。简言之，可移植性最高的项目往往“开箱可用”，而可移植性最低的构建则需要你跳过一个个难缠的框框，配置平台相关的路径以定位构建工具。在我们展示如何实现构建移植性之前，先浏览一些不同种类的构建可移植性。

11.1.1.1. 不可移植构建

缺少可移植性正是所有构建工具试图防止的——然而，任何工具都能被配置成不可移植（即使Maven）。一个不可移植的项目只有在一组特定的环境和标准（比如，你的本地机器）下才能构建。除非你自己一个人工作，不打算将你的应用部署到其它机器上，否则最好完全避免不可移植性。一个不可移植的构建只能在单独的机器上运行，是“一次性的”。Maven设计提供了使用profile自定义构建的能力，阻止不可移植的构建。

当一个新的开发人员得到不可移植项目的源码的时候，如果不重写大部分的构建脚本，他就不能构建这个项目。

11.1.1.2. 环境可移植性

如果一个构建有一种机制，能针对不同的环境有特定的行为和配置，那么我们就说该构建具有环境可移植性。例如，一个项目在测试环境中包含一个对于测试数据库的引用，而在产品环境中则引用了产品数据库，那么该项目的构建是环境可移植的。这很有可能

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

是因为该构建针对不同的环境有不同的属性组。当你转移到一个不同的环境中，该环境未被定义，也没有为其创建profile，那么项目将不能工作。因此，该项目也只是在已定义的环境中可移植。

当一个新的开发人员得到环境可移植项目的源码，他们必须在已定义的环境中运行此构建，否则就需要创建自定义的环境才能构建此项目。

11.1.1.3. 组织（内部）可移植性

这一层可移植性的中心是一个项目可能需要访问一些内部资源如源码控制系统或者内部维护的Maven仓库。大公司的项目可能依赖于一个只对内部开发人员可用的数据库，一个开源项目可能需要一个特定级别的证书来发布web站点，以及将构建的产品发布到公共仓库。

如果你试图在内网外部（例如，在公司的防火墙外面）从零开始构建一个内部项目，构建会失败。它失败的原因可能是一些必须的自定义插件不可用，或者项目的依赖找不到，因为你没有适当的证书从一个自定义的远程仓库获取依赖。这样的项目只在一个组织内部的环境中拥有可移植性。

11.1.1.4. 广泛（全局）可移植性

任何人都可以下载具有广泛可移植性项目的源码，不用为特定的环境自定义构建就能进行编译，安装。这是最高级别的可移植性；构建这样的项目不需要做任何额外的工作。该级别的可移植性对开源项目尤为重要，因为开源项目的潜在贡献者需要能很方便的下载源码进行构建。

任何一个开发者都可以下载具有广泛可移植性项目的源码。

11.1.2. 选择一个适当级别的可移植性

很显然，你需要避免创建出最坏的情况：不可移植构建。你可能不幸需要在这样的一个组织工作或学习：其核心应用的构建是不可移植的。在这样的组织下，没有特定的人员或机器的帮助，你就不可能部署一个应用。如果不和那个维护该不可移植构建的人协调，很难引入新的项目依赖或变化。当个人或小组需要控制项目如何以及何时构建部署的时候，不可移植构建就会由于一些政治环境因素高速增长。“我们该如何构建该系统？哦，我们需要找到Jack，让他帮我们构建，没有其他人能将其部署到产品环境中”这是一种非常危险的情形，该情形比你想象的普遍得多。如果你为这样的一个组织工作，Maven和Maven profile能帮你脱离困境。

与之完全相反的可移植性范围是广泛可移植性构建。总的来说广泛可移植性构建是最难达到的构建系统。这样的构建严格要求你依赖的项目和工具是免费分发的，在公共环境中可用。很多商业软件包可能被排除在这种可移植性最高的构建外面，因为只有在接受某个许可证后你才能下载它们。广泛可移植性也限制它的依赖能以Maven构件的形式分发。例如，如果你依赖于Oracle JDBC驱动，你的用户就需要手工下载安装它们；这

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

就不是广泛可移植性，因为你必须为那些有兴趣构建你应用的人发布一组关于创建环境的指令。而另一方面，你可以使用一个在公共Maven仓库中可用的JDBC驱动如MySQL或者HSQLDB。

如之前叙述的那样，开源项目从拥有最广泛的可移植性构建获益。广泛可移植性构建降低了为开源项目做贡献的低效性。在一个开源项目（如Maven）中，有两个单独的组：最终用户和开发者。当一个项目的最终用户决定为项目贡献一个补丁，它们就需要从使用构建输出过渡到运行一个构建。它们首先需要成为开发者，而如果很难学会如何构建一个项目，这就成为了最终用户花时间为项目做贡献的妨碍因素。在广泛可移植项目中，最终用户不需要遵循一组神秘的构建指令来开始成为开发者。他们可以签出源码，修改源码，构建，然后提交贡献，这个过程不需要问谁要求帮助建立构建环境。当为开源项目贡献源码的成本更低的时候，你就会看到源码贡献的增长，特别是一些不经意的贡献可能对项目的成功和项目的失败造成很大的影响。在一组广泛的开源项目中使用Maven的一个附加作用就是它使得开发者为不同的开源项目贡献源码变得更加容易。

11.2. 通过Maven Profiles实现可移植性

Maven中的profile是一组可选的配置，可以用来设置或者覆盖配置默认值。有了profile，你就可以为不同的环境定制构建。profile可以在pom.xml中配置，并给定一个id。然后你就可以在运行Maven的时候使用的命令行标记告诉Maven运行特定profile中的目标。以下pom.xml使用production profile覆盖了默认的Compiler插件设置。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

例 11.1. 使用一个Maven Profile覆盖Compiler插件设置

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                      http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.sonatype.mavenbook</groupId>
  <artifactId>simple</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>simple</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
  <profiles>#
    <profile>
      <id>production</id>#
      <build>#
        <plugins>
          <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <configuration>
              <debug>false</debug>#
              <optimize>true</optimize>
            </configuration>
          </plugin>
        </plugins>
      </build>
    </profile>
  </profiles>
</project>
```

本例中，我们添加了一个名为production的profile，它覆盖了Maven Compiler插件的默认配置，现在仔细看一下这个profile的语法。

❶ pom.xml中的profiles元素，它包含了一个或者多个profile元素。由于profile覆盖了pom.xml中的默认设置，profiles通常是pom.xml中的最后一个元素。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

❷ 每个profile必须要有一个id元素。这个id元素包含的名字将在命令行调用profile时被用到。我们可以通过传给Maven一个-P<profile_id>参数来调用profile。

❸ 一个profile元素可以包含很多其它元素，只要这些元素可以出现在POM XML文档的project元素下面。本例中，我们覆盖了Compiler插件的行为，因此必须覆盖插件配置，该配置通常位于一个build和一个plugins元素下面。

❹ 我们覆盖了Maven Compiler插件的配置。确保通过production profile产生的字节码不会包含调试信息，并且字节码会被编译器优化。

要使用production profile来运行mvn install，你需要在命令行传入-Pproduction参数。要验证production profile覆盖了默认的Compiler插件配置，可以像这样以开启调试输出(-X) 的方式运行Maven。

```
~/examples/profile $ mvn clean install -Pproduction -X
... (omitting debugging output) ...
[DEBUG] Configuring mojo 'o.a.m.plugins:maven-compiler-plugin:2.0.2:testCompile'
[DEBUG]   (f) basedir = ~\examples\profile
[DEBUG]   (f) buildDirectory = ~\examples\profile\target
...
[DEBUG]   (f) compilerId = javac
[DEBUG]   (f) debug = false
[DEBUG]   (f) failOnError = true
[DEBUG]   (f) fork = false
[DEBUG]   (f) optimize = true
[DEBUG]   (f) outputDirectory = \
          ~\svnw\sonatype\examples\profile\target\test-classes
[DEBUG]   (f) outputFileName = simple-1.0-SNAPSHOT
[DEBUG]   (f) showDeprecation = false
[DEBUG]   (f) showWarnings = false
[DEBUG]   (f) staleMillis = 0
[DEBUG]   (f) verbose = false
[DEBUG] -- end configuration --
... (omitting debugging output) ...
```

Maven的调试输出体现了production profile下Compiler插件的配置。可以看到，debug被设置成false，optimize设置成true。

11.2.1. 覆盖一个项目对象模型

虽然前面的样例展示了如何覆盖一个Maven插件的默认配置属性，你仍然没有确切知道Maven profile能覆盖什么。简单的回答这个问题，Maven profile可以覆盖几乎所有pom.xml中的配置。Maven POM包含一个名为profiles的元素，它包含了项目的替代配置，在这个元素下面，每个profile元素定义了一个单独的profile。每个profile必须要有一个id，除此之外，它可以包含几乎所有你能在project下看到的元素。以下的XML文档展示了一个profile允许覆盖的所有的元素。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

例 11.2. Profile中允许出现的元素

```
<project>
  <profiles>
    <profile>
      <build>
        <defaultGoal>...</defaultGoal>
        <finalName>...</finalName>
        <resources>...</resources>
        <testResources>...</testResources>
        <plugins>...</plugins>
      </build>
      <reporting>...</reporting>
      <modules>...</modules>
      <dependencies>...</dependencies>
      <dependencyManagement>...</dependencyManagement>
      <distributionManagement>...</distributionManagement>
      <repositories>...</repositories>
      <pluginRepositories>...</pluginRepositories>
      <properties>...</properties>
    </profile>
  </profiles>
</project>
```

一个Profile可以覆盖项目构件的最终名称，项目依赖，插件配置以影响构建行为。Profile还可以覆盖分发配置；例如，如果你通过一个staging profile发布一个构件到staging服务器上，你就可以创建一个profile然后在里面定义distributionManagement元素。

11.3. 激活Profile

在之前的小节中我们介绍了如何为一个特定的目标环境创建一个profile以覆盖默认行为。前面的例子中默认的构建是针对开发环境设计的，而production profile的存在就是为了为产品环境提供配置。当你需要基于一些变量如操作系统和JDK版本来进行配置的时候怎么做呢？Maven提供了一种针对不同环境参数“激活”一个profile的方式，这就叫做profile激活。

看如下的例子，假设我们拥有一个Java类库，它有一个特定的功能只有在Java 6下才可用，它需要使用定义在JSR-223[1]中的脚本引擎。你已经将那部分处理脚本的类库分离到了一个单独的Maven模块中，并且希望运行Java 5的人们能构建整个项目，而不去构建那部分针对Java 6的扩展类库。你可以使用一个Maven profile，只有构建在Java 6

[1] http://jcp.org/en/jsr/detail?id=223

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

JDK上运行的时候才将脚本扩展模块添加到构建中。首先，让我们看一下这个项目的目录布局，以及我们希望开发者如何构建该系统。

当有人使用Java 6 JDK运行mvn install的时候，你希望构建包含simple-script模块，而它们使用Java 5的时候，你希望跳过simple-script模块的构建。假如在Java 5下不能够跳过simple-script模块的构建，构建会失败因为Java 5的classpath中没有ScriptEngine。让我们看一下该类库项目的pom.xml：

例 11.3. 使用Profile激活动态包含子模块

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                             http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.sonatype.mavenbook</groupId>
  <artifactId>simple</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>simple</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
  <profiles>
    <profile>
      <id>jdk16</id>
      <activation>#
        <jdk>1.6</jdk>
      </activation>
      <modules>#
        <module>simple-script</module>
      </modules>
    </profile>
  </profiles>
</project>
```

如果你在Java 1.6下运行mvn install，你会看到Maven下行到simple-script子目录构建simple-script项目。如果你在Java 1.5上运行mvn install，Maven就不会去构建simple-script子模块。让我们详细看一下激活配置：

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

❶ activation元素列出了所有激活profile需要的条件。该例中，我们的配置为，当Java版本以“1.6”开头的时候profile会被激活。这包括“1.6.0_03”，“1.6.0_02”以及所有其它以“1.6”开头的字符串。激活参数不局限于Java版本，要了解激活参数的完整列表，请看激活配置。

❷ 在这个profile中我们添加了模块simple-script。这么做会让Maven从simple-script/子目录中寻找一个pom.xml文件。

11.3.1. 激活配置

激活配置元素下可以包含一个或者多个选择器：包含JDK版本，操作系统参数，文件，以及属性。当所有标准都被满足的时候一个profile才会被激活。例如，一个profile可以要求操作系统家族为Windoes，JDK版本为1.4，那么该profile只有当构建在Windows机器上的Java 1.4上运行的时候才会被激活。如果该profile被激活，那么它定义的所有配置都会覆盖原来POM中对应层次的元素，就像使用命令行参数-P引入该profile一样。下面的例子展示了一个profile，它通过一个十分复杂的配置组合激活，包括操作系统参数，属性，以及JDK版本。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

例 11.4. Profile激活参数：JDK版本，操作系统参数，以及属性

```
<project>
  ...
  <profiles>
    <profile>
      <id>dev</id>
      <activation>
        <activeByDefault>false</activeByDefault>#
        <jdk>1.5</jdk>#
        <os>
          <name>Windows XP</name>#
          <family>Windows</family>
          <arch>x86</arch>
          <version>5.1.2600</version>
        </os>
        <property>
          <name>mavenVersion</name>#
          <value>2.0.5</value>
        </property>
        <file>
          <exists>file2.properties</exists>#
          <missing>file1.properties</missing>
        </file>
      </activation>
      ...
    </profile>
  </profiles>
</project>
```

上例定义了一组狭小的激活参数集合。让我们仔细看一下每个激活配置：

❶ activeByDefault元素控制一个profile是否默认被激活。

❷ 该profile只有当JDK版本以“1.5”开头的时候才被激活。这包含“1.5.0_01”，“1.5.1”等。

❸ 该profile针对于一个特定的Windowx XP版本，32位平台上的5.1.2600版本。如果你的项目使用本地插件来构建一个C程序，你可能会发现自己正为特定的平台编写项目。

❹ property元素告诉Maven，当mavenVersion属性的值被设置成2.0.5的时候才激活profile。mavenVersion是一个在所有Maven构建中可用的隐式属性。

❺ file元素告诉我们只有当某些文件存在（或者不存在）的时候才激活profile。该例中的dev profile只有在项目基础目录中存在file2.properties文件，并且不存在file1.properties文件的时候才被激活。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

11.3.2. 通过属性缺失激活

你可以基于一个属性如environment.type的值来激活一个profile。当environment.type等于dev的时候激活development profile，或者当environment.type等于prod的时候激活production profile。你也可以通过一个属性的缺失来激活一个profile。下面的配置中，只有在Maven运行过程中属性environment.type不存在profile才被激活。

例 11.5. 在属性缺失的情况下激活Profile

```
<project>
  ...
  <profiles>
    <profile>
      <id>development</id>
      <activation>
        <property>
          <name>!environment.type</name>
        </property>
      </activation>
    </profile>
  </profiles>
</project>
```

注意属性名称前面的惊叹号。惊叹号通常表示“否定”的意思。当没有设置${environment.type}属性的时候，这个profile被激活。

11.4. 外部Profile

如果你开始大量使用Maven profile，你会希望将profile从POM中分离，使用一个单独的文件如profiles.xml。你可以混合使用定义在pom.xml中和外部profiles.xml文件中的profile。只需要将profiles元素放到/usr/local/hudson/hudson-home/jobs/maven-guide-zh-to-production/workspace/content-zh目录下的profiles.xml文件中，然后照常运行Maven就可以。profiles.xml文件的大概内容如下:

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

例 11.6. 将profile放到一个profiles.xml文件中

```
<profiles>
   <profile>
     <id>development</id>
     <build>
       <plugins>
         <plugin>
           <groupId>org.apache.maven.plugins</groupId>
           <artifactId>maven-compiler-plugin</artifactId>
           <configuration>
             <debug>true</debug>
             <optimize>false</optimize>
           </configuration>
         </plugin>
       </plugins>
     </build>
   </profile>
   <profile>
     <id>production</id>
     <build>
       <plugins>
         <plugin>
           <groupId>org.apache.maven.plugins</groupId>
           <artifactId>maven-compiler-plugin</artifactId>
           <configuration>
             <debug>false</debug>
             <optimize>true</optimize>
           </configuration>
         </plugin>
       </plugins>
     </build>
   </profile>
 </profiles>
```

你可以发现一旦你的Profile变得很大，再让你管理pom.xml会变得很困难，或者说只是因为你觉得将profiles.xml文件从中pom.xml分离出来是一种更干净的方式。不管怎样，调用定义在pom.xml中的profile和调用定义在profiles.xml中profile的方式是一样的。

11.5. Settings Profile

当一个项目需要为特定的环境自定义构建设置的时候，profile很有用，但是为什么一定要为所有Maven项目覆盖构建设置呢？比如为每个Maven构建添加一个需要访问的内部仓库。你可以使用一个settings profile做这件事情。项目profile关心

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

于覆盖某个项目的配置，而settings profile可以应用到所有你使用Maven构建的项目。你可以在两个地方定义settings profile：定义在~/.m2/settings.xml中的用户特定settings profile，或者定义在/usr/local/maven/conf/settings.xml中的全局settings profile。这里是一个定义在~/.m2/settings.xml中的settings profile的例子，它为所有的构建设置了一些用户特定的配置属性。如下settings.xml文件为用户tobrien定义：

例 11.7. 定义用户特定的Setting Profile (~/.m2/settings.xml)

```
<settings>
  <profiles>
    <profile>
      <id>dev</id>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-jar-plugin</artifactId>
        <executions>
           <execution>
              <goals>
                 <goal>sign</goal>
              </goals>
           </execution>
        </executions>
        <configuration>
           <keystore>/home/tobrien/java/keystore</keystore>
           <alias>tobrien</alias>
           <storepass>s3cr3tp@ssw0rd</storepass>
           <signedjar>
          /usr/local/hudson/hudson-home/jobs/maven-guide-zh-to-production/workspace/
           </signedjar>
           <verify>true</verify>
        </configuration>
      </plugin>
    </profile>
  </profiles>
</settings>
```

前面的例子是一个对于用户特定settings profile的真实用例。该样例中，当发布时为一个JAR文件签名的时候，设置用户特定的信息如密码和别名。你不会希望将这些配置参数存储到项目共享的pom.xml或者profiles.xml文件中，因为它们包含了一些不应该被公开的私人信息。

settings profile的缺点是它们可能会干扰项目可移植性。如果前面的例子是一个开源项目，如果一个新的开发者没有和其它开发者交流过并手工配置了一个settings profile，他将不能够为一个JAR签名。这样的情况下，为JAR签名的安全需求就与全局

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

可移植性构建产生了冲突。在大部分开源项目中，有一些任务需要安全证书：将一个构件发布到远程仓库，发布项目的web站点，或者为JAR文件签名。对于这些任务，可以达到的最高级别可移植性只能是组织可移植性。这些高安全性任务通常需要一些手工的profile安装和配置。

除了显式的使用-P命令行参数指定profile的名称。你还可以为所有构建的项目定义一个激活profile的列表。例如，如果你想要为每个项目激活定义在settings.xml中的dev profile，你可以在你的~/.m2/settings.xml文件中添加如下的设置：

例 11.8. 定义激活的Settings Profile

```
<settings>
  ...
  <activeProfiles>
    <activeProfile>dev</activeProfile>
  </activeProfiles>
</settings>
```

该设置只会激活settings profile，不会激活id匹配的项目profile。例如，如果你有一个项目，在pom.xml中定义了一个id为dev的profile，那个profile不会受你settings.xml中activeProfile设置的影响。activeProfile设置只对你settings.xml文件中定义的profile有效。

11.5.1. 全局Settings Profile

如同用户特定的settings profile一样，你也可以在/usr/local/maven/conf/settings.xml中定义一组全局profile。在这个配置文件中定义的profile对所有使用该Maven安装的用户可用。如果你正为一个特定的组织创建一个定制的Maven分发包，并且你想要确保每个Maven用户使用一组构建profile以确保内部可移植性，定义全局settings profile就十分有用。如果你需要添加自定义插件仓库，或者定义一组只在你组织内部可用的自定义插件，你就可以为你的用户分发一个内置了这些配置的Maven。配置全局settings profile和配置用户特定的settings profile的方法完全一样。

11.6. 列出活动的Profile

Maven profile可以通过pom.xml，profiles.xml，~/.m2/settings.xml，或者/usr/local/maven/conf/settings.xml定义。由于有四个层次，除了记住哪个文件中定义了哪个profile，没什么好的方式可以了解某个特定项目可用的profile。为了更方便的了解某个项目可用的profile，以及它们是在哪里定义的，Maven Help插件定义了一个目标，active-profiles，它能列出所有激活的profile，以及它们在哪里定义。你可以如下运行active-profiles目标：

```
$ mvn help:active-profiles
Active Profiles for Project 'My Project':
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛


```
The following profiles are active:

 - my-settings-profile (source: settings.xml)
 - my-external-profile (source: profiles.xml)
 - my-internal-profile (source: pom.xml)
```

11.7. 提示和技巧

Profile可以用来鼓励构建可移植性。如果你的构建需要为不同的平台做一些细微的自定义，或者如果你针对不同的目标平台生成不同的输出，项目profile就能增加构建可移植性。Settings profile通常会降低构建可移植性，因为它会添加一些必须在开发人员之间沟通的额外项目信息。下面的小节提供了一些如何在你项目中使用Maven profile的指导意见和想法。

11.7.1. 常见的环境

创建Maven项目profile的最核心动机之一就是为了提供环境特定的配置。在开发环境中，你可能想要生成带有调试信息的字节码，配置你的系统使用开发数据库实例。而在产品环境中，你可能会想要生成一个已签名的JAR，并且配置系统使用产品数据库。本章，我们使用dev和prod等标识符定义了很多环境。一种更简单的方式是，定义一些会被环境属性自动激活的profile，在你所有的项目中使用这些通用的环境属性。例如，如果每个项目都有一个development profile，当属性environment.type的值为dev时被激活，并且同样的项目还有production profile，当属性environment.type的值为prod时被激活。那么，你就可以在你的settings.xml中创建一个默认的profile，在你的开发机器上，总是将environment.type设置为dev。这样，所有定义了dev profile的项目都会由同样的系统变量激活。让我们看看如何完成这件事情，以下的~/.m2/settings.xml定义了一个profile，将environment.type属性设置成了dev。

例 11.9. ~/.m2/settings.xml 中定义一个设置了environment.type的默认profile，

```
<settings>
  <profiles>
    <profile>
      <activation>
        <activeByDefault>true</activeByDefault>
      </activation>
      <properties>
        <environment.type>dev</environment.type>
      </properties>
    </profile>
  </profiles>
</settings>
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

这意味着每次你在你机器上运行Maven的时候，该profile总会被激活，属性environment.type的值被设置为dev。之后你就可以使用这个属性来激活定义在某个如下项目pom.xml中的profile。让我们看一下如何在项目的pom.xml定义一个当属性environment.type的值为dev时被激活的profile。

例 11.10. 项目profile，当environment.type等于'dev'时被激活

```
<project>
  ...
  <profiles>
    <profile>
      <id>development</id>
      <activation>
        <property>
          <name>environment.type</name>
          <value>dev</value>
        </property>
      </activation>
      <properties>
        <database.driverClassName>com.mysql.jdbc.Driver</database.driverClassName>
        <database.url>
          jdbc:mysql://localhost:3306/app_dev
        </database.url>
        <database.user>development_user</database.user>
        <database.password>development_password</database.password>
      </properties>
    </profile>
    <profile>
      <id>production</id>
      <activation>
        <property>
          <name>environment.type</name>
          <value>prod</value>
        </property>
      </activation>
      <properties>
        <database.driverClassName>com.mysql.jdbc.Driver</database.driverClassName>
        <database.url>jdbc:mysql://master01:3306,slave01:3306/app_prod</database.ur
        <database.user>prod_user</database.user>
      </properties>
    </profile>
  </profiles>
</project>
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

该项目定义了一些属性如database.url和database.user，它们可能被用来配置一些定义在pom.xml中的Maven插件。有很多可用的插件可以用来操作数据库，运行SQL，而且还有如Maven Hibernate3之类的插件可以帮你在使用持久化框架的时候生成注解模型对象。这其中的一些插件，可以在pom.xml中被配置成使用这些属性。这些属性还可以被用来过滤资源。本例中，因为我们在~/.m2/settings.xml中定义了一个profile，设置environment.type为dev，在这台开发机器上运行Maven的时候，development profile就一直会被激活。做为选择，如果我们想要覆盖缺省值，我们可以在命令行设置这个属性值。如果我们需要激活production profile，我们可以如下运行Maven：

```
~/examples/profiles $ mvn install -Denvironment.type=prod
```

在命令行设置一个属性可以覆盖定义在~/.m2/settings.xml中的缺省值。我们可以仅仅定义一个id为"dev"的profile，然后直接使用-P命令行参数调用它，但是使用environment.type属性允许我们在编写其它项目的pom.xml时遵循这个标准。在你代码库中的每个项目都可以有一个profile，由定义在每个用户~/.m2/settings.xml中的项目的environment.type属性激活。这样，开发人员就能共享开发配置，而不用在不可移植的settings.xml文件中定义这些配置。

11.7.2. 安全保护

该最佳实践基于前面一小节。在项目profile，当environment.type等于'dev'时被激活中，production profile不包含database.password属性。我特地这么做是为了说明一个概念：将秘密信息放到你的用户特定的settings.xml文件中。如果你在一个大型组织中开发一个应用，那么很可能开发小组的大部分人不知道产品数据库的密码。如果一个组织的开发小组和运营小组有明确的界限的话，这种安全保护就很常见了。开发人员可以访问开发环境和staging环境，但是它们往往不被允许访问产品数据库。为什么要这么做有很多原因，尤其是当一个组织处理异常敏感的经济，情报，或者医疗信息的时候。在这样的情况下，产品环境构建只能有开发者领导或者产品运营小组成员执行。当它们使用environment.type的时候，它们会需要在settings.xml在中定义如下变量。

例 11.11. 在用户特定Settings Profile中存储秘密信息

```
<settings>
  <profiles>
    <profile>
      <activeByDefault>true</activeByDefault>
      <properties>
        <environment.type>prod</environment.type>
        <database.password>m1ss10nimp0ss1bl3</database.password>
      </properties>
    </profile>
  </profiles>
</settings>
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

这个用户定义了一个默认profile，将environment.type设置成prod，同时也设置了产品数据库密码。当项目构建的时候，production profile由environment.type属性激活，并且database.password属性也被填充。这样，你就可以将所有产品项目的配置放到项目的pom.xml中，而不用在那里配置访问产品数据库必要的秘密信息。

注意

秘密信息通常会和可移植性冲突，但这是合理的。你不会想公开你的秘密信息。

11.7.3. 平台分类器

假设你有一个类库，或者一个项目，它针对不同的平台有不同的输出。即使Java是平台无关的，但还是有一些时候，你会需要编写一些代码调用平台项目的本地代码。另一个可能性就是你编写了一些使用Maven Native插件编译的C代码，并且要基于构建平台生成已修饰的构件。你可以通过Maven Assembly插件或者Maven Jar插件设置一个分类器。以下的pom.xml使用了由操作系统参数激活的profile来生成已修饰的构件。要了解更多的关于Maven Assembly插件的信息，请参考第 12 章Maven套件。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

例 11.12. 使用由平台激活的Profile修饰构件

```
<project>
  ...
  <profiles>
    <profile>
      <id>windows</id>
      <activation>
        <os>
          <family>windows</family>
        </os>
      </activation>
      <build>
        <plugins>
          <plugin
            <artifactId>maven-jar-plugin</artifactId>
            <configuration>
              <classifier>win</classifier>
            </configuration>
          </plugin>
        </plugins>
      </build>
    </profile>
    <profile>
      <id>linux</id>
      <activation>
        <os>
          <family>unix</family>
        </os>
      </activation>
      <build>
        <plugins>
          <plugin>
            <artifactId>maven-jar-plugin</artifactId>
            <configuration>
              <classifier>linux</classifier>
            </configuration>
          </plugin>
        </plugins>
      </build>
    </profile>
  </profiles>
</project>
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

如果操作系统是Windows家族，该pom.xml就使用"-win"修饰这个JAR构建。如果操作系统是Unix家族，该构件就由"-linux"修饰。pom.xml成功的给构件添加修饰符，但是由于在两个profile中的Maven Jar插件的冗余配置，该文件变得有些累赘了。该样例可以被重写，如下，使用变量替换来减少冗余：

例 11.13. 使用由平台激活的Profile和变量替换修饰构件

```
<project>
  ...
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-jar-plugin</artifactId>
        <configuration>
          <classifier>${envClassifier}</classifier>
        </configuration>
      </plugin>
    </plugins>
  </build>
  ...
  <profiles>
    <profile>
      <id>windows</id>
      <activation>
        <os>
          <family>windows</family>
        </os>
      </activation>
      <properties>
        <envClassifier>win</envClassifier>
      </properties>
    </profile>
    <profile>
      <id>linux</id>
      <activation>
        <os>
          <family>unix</family>
        </os>
      </activation>
      <properties>
        <envClassifier>linux</envClassifier>
      </properties>
    </profile>
  </profiles>
</project>
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

在这个pom.xml中，每个profile不需要包含一个build元素来配置Jar插件。每个profile通过操作系统家族参数激活，并且设置envClassifier属性为win或者linux。这个envClassifier属性由缺省pom.xml的build元素引用，以为项目的JAR构建添加分类器。JAR构件将会被命名为${finalName}-${envClassifier}.jar，并且需要按如下的依赖语法引用。

例 11.14. 依赖于一个已修饰的构件

```
<dependency>
      <groupId>com.mycompany</groupId>
      <artifactId>my-project</artifactId>
      <version>1.0</version>
      <classifier>linux</classifier>
    </dependency>
```

11.8. 小结

如果正确的使用profile，它可以为不同的平台很容易的自定义构建。如果你构建中的一些东西需要定义一个平台特定的路径，如应用程序服务器，你可以将这些配置点放到profile中，然后由操作系统参数激活。如果你有一个项目需要为不同的环境生成不同的构件，你可以为不同的环境和平台自定义profile特定的插件行为，从而自定义构建行为。使用profile，构建可以变得容易移植，没有必要为一个新的环境重写你的构建逻辑，只需要重写那些需要变化的配置，共享那些可以被共享的配置。

第 12 章 Maven套件

12.1. 简介

Maven提供了可以用来创建大多数常见归档类型的插件，这些归档类型中的大部分可以由其它项目作为依赖使用。这样的插件有JAR, WAR, EJB, 以及 EAR插件。如在第 10 章构建生命周期中讨论的那样，这些插件对应于不同的项目打包类型，每种类型的构建过程都有微小的差别。虽然Maven有插件及自定义生命周期来支持标准的打包类型，但还是有些时候你需要创建一个自定义格式的归档文件或目录。这样的归档叫做Maven套件(Assembly)。

在很多情况下你需要为你的项目构建自定义的归档。最常见的可能就是项目分发。单词“分发(distribution)”意思是，根据项目将会如何被使用，为不同的人（或项目）提供不同的东西。本质上来说，这些归档是为用户提供一种便捷的方式来安装，否则用户就必须使用项目的发布版本。一些情况下，这可能是构建一个带有应用服务器（如Jetty）的web应用。还有一些情况，可能是包裹了项目API文档，源码，和已编译字节码的JAR文件。当你构建项目的最终分发包的时候，Maven套件就十分有用了。以第 16 章仓库管理器中介绍的Nexus为例，它是一个大型的包含了很多模块Maven项目的产品，而你从Sonatype下载的最终归档就是一个Maven套件。

大多数情况下，Assembly插件十分适用于构建项目的分发包。然而，套件不一定就要是分发包；套件的目的是能让用户灵活的构建任意类型的自定义归档文件。本质上说，套件是为了弥补由项目打包类型所提供的标准归档格式的不足。当然，你可以完全写一个Maven插件来帮助生成你的自定义归档格式，同时创建生命周期映射和构件处理配置来告诉Maven如何部署。但大部分情况下，有了Assembly插件，这就完全没有必要了。该插件为创建自定义归档格式提供了普遍的支持，你不需要花很多时间来编写Maven插件代码。

12.2. Assembly基础

在我们进一步讨论之前，最好先花几分钟来讲一下Assembly插件的两个主要目标：`assembly:assembly`，和`single` mojo。我用不同的方式列出这两个目标，是因为这样可以很好的体现它们不同的使用方式。`assembly:assembly`目标被设计成直接从命令行调用，它永远不应该被绑定到生命周期阶段。反之，`single` mojo被设计成作为你每日构建的一部分，应该被绑定到项目生命周期的某个阶段。

有这种区别的主要理由是，`assembly:assembly`目标在Maven术语中是一个聚合mojo；意思是，无论有多少个模块正被构建，该mojo在一个构建中最多被运行一次。它从根项目的中提取配置——通常是顶层POM或者命令行目录的POM。而在绑定到生命周期后，一个

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

聚合mojo就会有很多讨厌的副作用。它会强迫package生命周期阶段提前运行，从而造成在一次构建中package阶段被执行了两次。

由于assembly:assembly目标是一个聚合mojo，在多模块构建中它造成了一些问题，它应当在命令行中作为单独的mojo被调用。永远不要将assembly:assembly目标绑定到生命周期阶段中。assembly:assembly是Assembly插件最原始的目标，不是被设计成用作项目标准构建过程的一部分的。但很显然，为一个项目生成套件归档合情合理，因此开发了single mojo。该mojo假设它被绑定到了构建过程的正确部分，因此它能访问它在大型模块Maven项目生命周期中执行需要的项目文件及构件。在一个多模块环境中，它会执行很多次，因为它被绑定到了不同模块的POM上。不像assembly:assembly，single从来不会强制在它之前执行另外一个生命周期阶段。

除了上述的两项，Assembly插件还提供了很多其它目标；然后，讨论这些mojo超出了本章范围，主要是由于它们是为一切奇怪或者过时的用例服务的，基本上不会被用到。只要可能，你都应该坚持在命令行生成套件时使用assembly:assembly，在绑定到生命周期阶段生成套件时使用single。

12.2.1. 预定义的套件描述符

虽然很多人选择创建他们自己的归档解决方案——称之为套件描述符——但这不是必须的。Assembly插件为一些常用的归档类型提供了内置的描述符，你可以不写 一行配置就马上使用。以下的套件描述符是由Maven Assembly插件预定义的：

bin
: 假设一个项目构建了一个jar作为它的主构件，bin 描述符用来包裹该主构件和项目的LICENSE, README, 和NOTICE文件。我们可以认为这是一个完全自包含项目的最可能小的二进制分发包。

jar-with-dependencies
: jar-with-dependencies描述符构建一个带有主项目jar文件和所有项目运行时依赖未解开内容的JAR归档文件。外加上适当的Main-Class Manifest条目（在下面的“插件配置”讨论），该描述符可以为你的项目生成一个自包含的，可运行的jar，即使该项目含有依赖。

project
: project描述符会简单的将你文件系统或者版本控制中的项目目录结构整个的归档。当然，target目录会被忽略，目录中的版本控制元数据文件如.svn和CVS目录也会被忽略。基本上，该描述符的目的是创建一个解开后就立刻能由Maven构建的归档。

src
: src描述符生成一个包含你项目源码，pom.xml文件，以及项目根目录中所有LICENSE，README，和NOTICE文件的归档。它类似于project描述符，大部分情

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

况下能生成一个可以被Maven构建的归档。然而，由于它假设所有的源文件和资源文件都位于标准的src目录下，它就可能遗漏那些非标准的目录和文件，而这些文件往往对构建起着关键作用。

12.2.2. 构建一个套件Building an Assembly

Assembly插件可以以两种方式运行：你可以直接在命令行调用，或者你可以通过绑定到生命周期阶段将其配置成标准构建过程的一部分。直接调用自有它的用处，主要是为了那些一次性，不属于你项目核心分发包的套件。大部分情况下，你可能以标准项目构建过程一部分的形式生成套件。这么做，不管你的项目被安装还是部署，都能包含你自定义的套件，从而它们也一直是对用户可用的。

作为一个直接调用Assembly插件的例子，考虑你想要将你的项目复制一份给别人让他们也能从源码构建。你需要同时包含源码，而不仅仅是一个最终的产品。而且你只是偶尔需要这么做，因此往你的POM中添加配置显得没有必要。你可以使用如下的命令：

```
$ mvn -DdescriptorId=project assembly:single
...
[INFO] [assembly:single]
[INFO] Building tar : /Users/~/mvn-examples-1.0/assemblies/direct-invocation/\
                      target/direct-invocation-1.0-SNAPSHOT-project.tar.gz
[INFO] Building tar : /Users/~/mvn-examples-1.0/assemblies/direct-invocation/\
                      target/direct-invocation-1.0-SNAPSHOT-project.tar.bz2
[INFO] Building zip: /Users/~/mvn-examples-1.0/assemblies/direct-invocation/\
                      target/direct-invocation-1.0-SNAPSHOT-project.zip
...
```

假如你想要从你的项目生成一个可运行的JAR。如果你的项目完全是自包含的，没有任何依赖，那么使用JAR插件的archive配置就能做到。但是，大部分项目都有依赖，而且这些依赖需要被引入到可运行JAR文件中。这种情况下，每次JAR被安装或部署的时候你都要确认可运行JAR包含了这些依赖。

假设该项目的main类是org.sonatype.mavenbook.App，如下的POM配置将创建一个可运行的JAR：

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/mave

  <modelVersion>4.0.0</modelVersion>
  <groupId>org.sonatype.mavenbook.assemblies</groupId>
  <artifactId>executable-jar</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  <name>Assemblies Executable Jar Example</name>
  <url>http://sonatype.com/book</url>
  <dependencies>
    <dependency>
      <groupId>commons-lang</groupId>
      <artifactId>commons-lang</artifactId>
      <version>2.4</version>
    </dependency>
  </dependencies>
 <build>
    <plugins>
      <plugin>
        <artifactId>maven-assembly-plugin</artifactId>
        <version>2.2-beta-2</version>
        <executions>
          <execution>
            <id>create-executable-jar</id>
            <phase>package</phase>
            <goals>
              <goal>single</goal>
            </goals>
            <configuration>
              <descriptorRefs>
                <descriptorRef>
                  jar-with-dependencies
                </descriptorRef>
              </descriptorRefs>
              <archive>
                <manifest>
                  <mainClass>org.sonatype.mavenbook.App</mainClass>
                </manifest>
              </archive>
           </configuration>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

以上配置中有两点我们要注意。首先，我们不再像上次那样使用descriptorId参数，这里我们使用descriptorRefs配置。这样可以允许在同一次Assembly插件执行过程中使用多个套件类型，同时还能保持不要引入太多额外的配置。第二，configuration下的archive元素设置最终JAR的Main-Class manifest属性。该配置在所有创建JAR文件的插件中都很常见，如由默认打包类型使用的JAR插件。

现在，你可以通过执行mvn package生成可运行的JAR文件。之后，我们看一下target目录的内容，验证一下可运行JAR已经被生成了。最后，为了证明我们真的有可运行JAR了，那就运行一下吧。

```
$ mvn package
... (output omitted) ...
[INFO] [jar:jar]
[INFO] Building jar: /Users/~/mvn-examples-1.0/assemblies/executable-jar/target/\
                     executable-jar-1.0-SNAPSHOT.jar
[INFO] [assembly:single {execution: create-executable-jar}]
[INFO] Processing DependencySet (output=)
[INFO] Building jar: /Users/~/mvn-examples-1.0/assemblies/executable-jar/target/\
                     executable-jar-1.0-SNAPSHOT-jar-with-dependencies.jar
... (output omitted) ...
$ ls -1 target
... (output omitted) ...
executable-jar-1.0-SNAPSHOT-jar-with-dependencies.jar
executable-jar-1.0-SNAPSHOT.jar
... (output omitted) ...
$ java -jar \
        target/executable-jar-1.0-SNAPSHOT-jar-with-dependencies.jar
Hello, World!
```

从以上的输出你可以看到，标准的项目构建现在生成了一个新的额外的构件。这个新的构件有一个名为jar-with-dependencies的分类器。最后，我们验证了这个新的JAR是可运行的，运行这个JAR会产生我们期望的输出“Hello, World!”。

12.2.3. 套件作为依赖

当你将生成套件作为你标准构建过程的一部分的时候，这些套件归档会被附着到你项目的主构件上。意思是，当它们会随着主构件的安装和部署一起被安装或部署，而且它们会以差不多相同的方式被解析。每个套件构件会被给予同样的基本坐标（groupId, artifactId, 和 version）。但是，由于这些构件是附件，在Maven中意思是，它们是基于主项目构建的某些方面派生出来的。这里是一些例子，source套件包含了项目的原始输入，jar-with-dependencies套件包含了项目的类及依赖。由于这种派生的性质，附属构件就能精确的规避一个项目，或一个构件的Maven需求。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

由于套件（通常）是附属构件，除了标准的构件坐标，它们还必须拥有一个分类器来和主构件加以区分。默认情况下，该分类器就是套件描述符的定义符。当如上例使用内置的套件描述符的时候，套件描述符的定义符和descriptorRef中对应使用的定义符是一样的。

在你伴随着项目的主构件部署了一个套件后，如何才能在其它项目中使用这个套件呢？答案很简单。回忆一下第 3.5.3 节 “Maven坐标 (Coordinates)”和第 9.5.1 节 “坐标详解”中讨论的项目依赖，一个项目依赖于另外一个项目的时候，它使用项目坐标四个元素的组合：groupId, artifactId, version, 和 packaging。在第 11.7.3 节 “平台分类器”中，同一个项目的构件有针对多个平台的变种，该项目设置了值为win或linux的classifier元素，以为目标平台选择合适的依赖构件。使用项目的坐标加上套件被安装或部署时的分类器，我们就可以将套件设为依赖。如果套件不是一个JAR归档，我们还需要声明其类型。

12.2.4. 通过套件依赖组装套件

起这么一个使人困惑的名字是怎么回事？让我们建立这样一个能解释组装套件(assembling assemblies)概念的场景。假设你想要创建一个归档，它包含了一些项目套件。再假设你有一个多模块构建，并且想要部署一个包含很多相关项目套件的套件。在本节的例子中，我们要为一组通常一起被使用的项目创建一个“可构建”的项目目录。为了简便，我们要重用前面讨论的两个project和jar-with-dependencies内置套件描述符。在这个特定的例子中，假设每个项目在主要的JAR构件之外还创建project套件。假设这个多模块构建中每个项目都使用project descriptorRef绑定到package阶段的single目标。每个模块都会从顶层的pom.xml中继承该配置，其pluginManagement元素如例 12.2 “在顶层POM中配置项目套件”所示：

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

例 12.2. 在顶层POM中配置项目套件

```
<project>
  ...
  <build>
    <pluginManagement>
      <plugins>
        <plugin>
          <artifactId>maven-assembly-plugin</artifactId>
          <version>2.2-beta-2</version>
          <executions>
            <execution>
              <id>create-project-bundle</id>
              <phase>package</phase>
              <goals>
                <goal>single</goal>
              </goals>
              <configuration>
                <descriptorRefs>
                  <descriptorRef>project</descriptorRef>
                </descriptorRefs>
              </configuration>
            </execution>
          </executions>
        </plugin>
      </plugins>
    </pluginManagement>
  </build>
  ...
</project>
```

每个项目的POM都使用很小的一部分插件声明引用了例 12.2 “在顶层POM中配置项目套件”中的插件配置，如例 12.3 “在子项目中激活Assembly插件配置”所示：

例 12.3. 在子项目中激活Assembly插件配置

```
<build>
  <plugins>
    <plugin>
      <artifactId>maven-assembly-plugin</artifactId>
    </plugin>
  </plugins>
</build>
```

要生成一组项目套件，从顶层目录运行mvn install。你会看到Maven使用分类器将套件安装到你的本地仓库。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛


```
$ mvn install
...
[INFO] Installing ~/mvn-examples-1.0/assemblies/as-dependencies/project-parent/\
                  second-project/target/second-project-1.0-SNAPSHOT-project.tar.gz
       ~/.m2/repository/org/sonatype/mavenbook/assemblies/second-project/1.0-SNAPSHO
                  second-project-1.0-SNAPSHOT-project.tar.gz
...
[INFO] Installing ~/mvn-examples-1.0/assemblies/as-dependencies/project-parent/\
                  second-project/target/second-project-1.0-SNAPSHOT-project.tar.bz2
       ~/.m2/repository/org/sonatype/mavenbook/assemblies/second-project/1.0-SNAPSHO
                  second-project-1.0-SNAPSHOT-project.tar.bz2
...
[INFO] Installing ~/mvn-examples-1.0/assemblies/as-dependencies/project-parent/\
                  second-project/target/second-project-1.0-SNAPSHOT-project.zip to
       ~/.m2/repository/org/sonatype/mavenbook/assemblies/second-project/1.0-SNAPSHO
                  second-project-1.0-SNAPSHOT-project.zip
...
```

当你运行install的时候，Maven会将每个项目的主构件和每个套件复制到你的本地Maven仓库中。现在本地的所有其它项目都可以使用依赖引用这些构件。如果你最终的目的是创建一个包含多个多模块项目构件的软件包，你可以创建一个项目，它以依赖的形式引入其它项目的套件。这个包裹项目（贴切的命名为project-bundle）负责创建一个最终的包裹套件。这个包裹项目的POM如例 12.4 “套件包裹项目的POM”所示：

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

```
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/mave
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.sonatype.mavenbook.assemblies</groupId>
  <artifactId>project-bundle</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>pom</packaging>
  <name>Assemblies-as-Dependencies Example Project Bundle</name>
  <url>http://sonatype.com/book</url>
  <dependencies>
    <dependency>
      <groupId>org.sonatype.mavenbook.assemblies</groupId>
      <artifactId>first-project</artifactId>
      <version>1.0-SNAPSHOT</version>
      <classifier>project</classifier>
      <type>zip</type>
    </dependency>
    <dependency>
      <groupId>org.sonatype.mavenbook.assemblies</groupId>
      <artifactId>second-project</artifactId>
      <version>1.0-SNAPSHOT</version>
      <classifier>project</classifier>
      <type>zip</type>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-assembly-plugin</artifactId>
        <version>2.2-beta-2</version>
        <executions>
          <execution>
            <id>bundle-project-sources</id>
            <phase>package</phase>
            <goals>
              <goal>single</goal>
            </goals>
            <configuration>
              <descriptorRefs>
                <descriptorRef>
                  jar-with-dependencies
                </descriptorRef>
              </descriptorRefs>
            </configuration>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

该包裹项目的POM引用了来自于first-project和second-project的两个套件。这个POM指定了一个值为project的分类器和一个值为zip的类型，而不是直接引用过了项目的主构件。这告诉Maven去解析project套件创建的ZIP归档。注意包裹项目生成了一个jar-with-dependencies套件。jar-with-dependencies并不创建一个非常简洁的套件，而是简单的创建一个包含所有依赖拆解内容的JAR文件。jar-with-dependencies实际上做的事情是：告诉Maven拿来所有依赖，将其拆解，然后创建一个包含当前项目所有输出的归档。在这个项目中，它的效果就是创建一个带有first-project和second-project套件拆解内容的JAR文件，

该样例展示了如何不用自定义套件描述符就能联合Maven Assembly插件的基本能力。它实现了创建一个包含多模块项目内容的单独归档这样一个目的。这个时候，jar-with-dependencies仅仅是一个存储格式，因此我们没必要指定Main-Class的manifest属性。我们只要正常的构建project-bundle项目就能得到套件。

```
$ mvn package
...
[INFO] [assembly:single {execution: bundle-project-sources}]
[INFO] Processing DependencySet (output=)
[INFO] Building jar: ~/downloads/mvn-examples-1.0/assemblies/as-dependencies/\
        project-bundle/target/project-bundle-1.0-SNAPSHOT-jar-with-dependencies.jar
```

要验证project-bundle套件是否包含了所有依赖套件的拆解内容，运行jar tf：

```
$ java tf \
  target/project-bundle-1.0-SNAPSHOT-jar-with-dependencies.jar
...
first-project-1.0-SNAPSHOT/pom.xml
first-project-1.0-SNAPSHOT/src/main/java/org/sonatype/mavenbook/App.java
first-project-1.0-SNAPSHOT/src/test/java/org/sonatype/mavenbook/AppTest.java
...
second-project-1.0-SNAPSHOT/pom.xml
second-project-1.0-SNAPSHOT/src/main/java/org/sonatype/mavenbook/App.java
second-project-1.0-SNAPSHOT/src/test/java/org/sonatype/mavenbook/AppTest.java
```

阅读本节内容之后，标题应该容易理解了。我们使用了一个包裹项目，它依赖于两个项目的套件，然后我们再根据这两个项目的套件装配出一个最终的套件。

12.3. 套件描述符概述

当第 12.2 节 “Assembly基础”中介绍的标准套件描述符不够的时候，你就需要自定义你自己的套件描述符。套件描述符是一个定义了套件结构和内容的XML文档。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

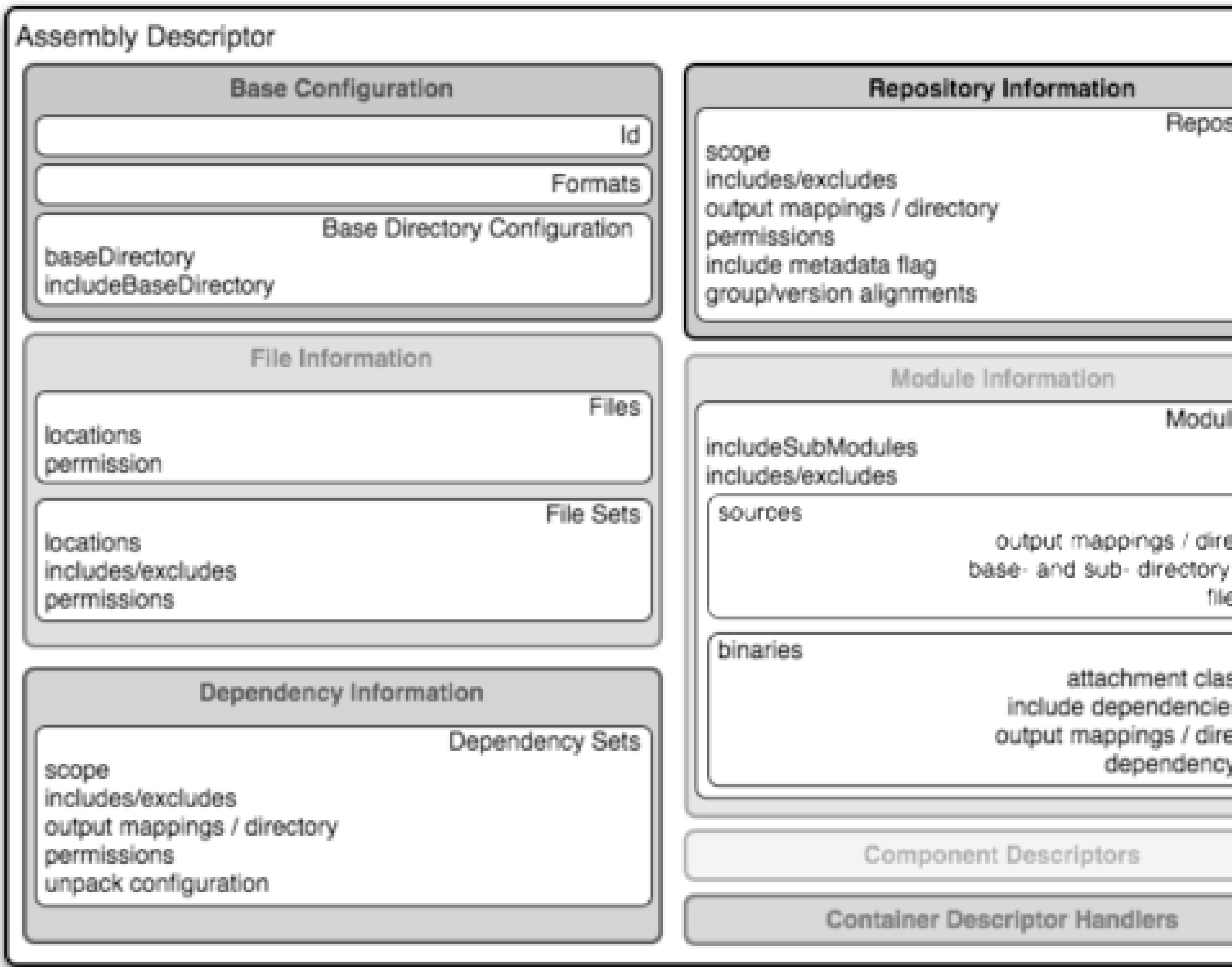


图 12.1. 套件描述符图解

套件描述符包含五个主要的小节，以及两个额外的小节：一个叫做组件描述符，用来指定标准的套件描述符片段，另一个用来指定自定义的文件处理类，以帮助管理套件生产过程。

基本配置
: 该小节包含了所有套件需要的信息，还有一些关于整个归档格式的额外配置选项，如所有归档项使用的基础路径。为了使套件描述符有效，你必须至少指定套件id，至少一种格式，以及至少一个如上所示的其它小节。

文件信息
: 套件描述符这个片段中的配置应用于文件系统中该项目目录结构中特定的文件。这个片段包含了两个主要部分：`files`和`fileSets`。你可以使用`files`和`fileSets`来控制套件中文件的权限，从套件中包含或者排除文件。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

依赖信息

几乎任意大小的所有的项目都依赖于其它项目。在创建分发归档的时候，项目依赖通常被包含在最终产品套件中。该小节管理依赖被包含在最终归档中的方式。该小节允许你依赖是否被拆解，是直接添加到lib/目录中，还是映射至新的文件名。该小节也允许你控制套件中依赖的权限，以及哪些依赖被包含在套件中。

仓库信息

有时，将构建一个项目必要的所有构件分隔开来很有用，它们是否是依赖构件，依赖构件的POM，或者甚至是一个项目本身POM的祖先（你的父POM，父POM的父POM，等等）。该小节允许你通过各种配置选项，在套件中包含一个或者多个构件仓库目录，然而Assembly插件没有能力引入在这些仓库中的插件构件。

模块信息

套件描述符的这一小节允许你在装配自定义归档的时候利用父子关系，以包含你项目模块的源文件，构件，和依赖。这是套件描述符中最复杂的小节，因为它允许你以两种方式操作模块和子模块：以一系列fileSets（通过sources小节）或者以一系列dependencySets（通过binaries小节）。

12.4. 套件描述符

本节作为一个套件描述符的教程，包含了一些开发自定义套件描述符的指南。Assembly插件是Maven世界中最大的插件之一，也是最灵活的插件之一。

12.4.1. 套件描述符中的属性引用

任何在第 13.2 节 “Maven属性”中讨论的属性都可以在一个套件描述符中引用。在套件描述符被Maven使用之前，它首先会根据POM及构建环境的信息修改一些对应的值。所有运行时POM中可修改的值，同样适用于套件描述符，包括POM属性，POM元素的值，系统属性，用户定义的属性，以及操作系统环境变量。

这一插值阶段的例外是outputDirectory，outputDirectoryMapping，和outputFileNameMapping元素。保持它们原始的值是为了能在解析构件特定的——或者组件特定的信息的时候，能得到正确的值。

12.4.2. 必须的套件信息

对于每个套件来说，有两个重要的信息是必须的：id，和一个要生成的归档格式的列表。实际情况中，至少还需要其它一小节描述符的内容——因为大部分归档格式如果没有可引入的文件，将一文不值，同时，如果没有一个id和至少一种格式，就不会产生任何归档。id首先被用到归档的文件名中，同时它还是归档构件分类器名称一部分。格式（format）字符串同时也控制archiver-component实例去创建最终的套件归档。所有套件描述符必须包含一个id和至少一个format：

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

例 12.5. 必须的套件描述符元素

```
<assembly>
  <id>bundle</id>
  <formats>
    <format>zip</format>
  </formats>
  ...
</assembly>
```

套件id可以是任意不包含空格的字符串。标准的实践是使用破折号（-）来分隔单词。如果你要创建一个拥有有趣的唯一打包结构（interesting unique package structure）的套件，那么其id就会是interesting-unique-package。同时，套件描述符还支持多种格式，允许你轻松创建.zip，.tar.gz,和.tar.bz2等熟悉分发归档。如果你没有找到你需要的归档格式，你也可以创建自定义的格式。自定义格式在第 12.5.8 节 “componentDescriptors和containerDescriptorHandlers”中讨论。Assembly插件原生支持多种归档格式，包括：

- jar
- zip
- tar
- bzip2
- gzip
- tar.gz
- tar.bz2
- rar
- war
- ear
- sar
- dir

id和format至关重要，因为它们将成为套件归档坐标的一部分。例 12.5 “必须的套件描述符元素”中的例子会创建一个打包类型为zip，分类器为bundle的套件构件。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

12.5. 控制一个套件的内容

理论上，只有id和format才是一个合法套件描述符必要的元素；然而，对于大多数归档来说，如果输出归档中没有包含一个文件，该归档就会失效。套件描述符中，有五个主要的部分来定义哪些文件被包含到套件中，他们是：files，fileSets，dependencySets，repositories，和moduleSets。为了最有效的探究这些部分，我们首先讨论最基本的部分：files。然后，我们接着讨论两个最常用的部分：fileSets和dependencySets。在你理解了它们如何工作之后，就能更轻松的理解repositories和moduleSets。

12.5.1. **Files** 元素

files元素是套件描述符中最简单的部分，它被设计成定义那些相对与你项目目录的路径。使用该元素，你就可以完全控制哪些文件被包含到你的套件中，它们如何被命名，以及它们在归档中的位置。

例 12.6. 使用files在一个套件中包含一个JAR文件

```
<assembly>
  ...
  <files>
    <file>
      <source>target/my-app-1.0.jar</source>
      <outputDirectory>lib</outputDirectory>
      <destName>my-app.jar</destName>
      <fileMode>0644</fileMode>
    </file>
  </files>
  ...
</assembly>
```

假设你要构建一个名为my-app的版本为1.0的项目，例 12.6 “使用files在一个套件中包含一个JAR文件”会将你项目的JAR文件包含到套件的lib/目录下，同时除去了文件名的版本部分，使最终的文件名为my-app.jar。该描述符会让这个JAR文件对于所有人可读，对于拥有者可写（这就是0664模式的意思，这里使用了Unix四位十进制数的权限标记）。要了解更多关于fileMode的值的格式的信息，请看维基百科的解释four-digit Octal notation[1]。

如果你知道所有需包含的文件的完整列表，你就可以使用file条目构建了一个十分复杂的套件。即使在构建开始之前你不知道这个完整的列表，你也可以使用一个自定义的

[1] http://en.wikipedia.org/wiki/File_system_permissions#Octal_notation_and_additional_permissions

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

Maven插件来发现这个列表，然后生成如上例中的套件描述符。files元素能给你提供细粒度的文件控制，包括每个文件的权限，位置及名称，但是在一个大型的归档中，为所有文件罗列file元素就比较繁琐了。因此大部分情况下，你需要使用fileSets针对一组文件进行操作。余下的四个文件包含配置元素，用来帮助你包含整个一组匹配某个特定标准的文件。

12.5.2. **FileSets** 元素

与files元素类似，fileSets应用于那些相对于你的项目结构有一个明确位置的文件。然而，和files元素不同的是，fileSets描述一组文件，这组文件由文件模式或者路径模式定义，判断文件在总体目录结构中的位置是否匹配该模式。最简单的fileSet仅仅指定文件所处的位置：

```
<assembly>
  ...
  <fileSets>
    <fileSet>
      <directory>src/main/java</directory>
    </fileSet>
  </fileSets>
  ...
</assembly>
```

这个文件集合仅仅包含src/main/java目录下的内容。它利用了很多该元素的默认设置，这里我简要的介绍一下。

首先，你会注意到我们没有指定匹配的文件集合应该位于套件中的什么位置。默认情况，目标输出目录（由outputDirectory指定）和源文件目录（在这里，是src/main/java）是一样的。此外，我们没有指定任何包含和排除模式。当它们为空的时候，fileSet假设会包含整个源文件目录，并伴随着一些重要的例外。这一规则的例外主要是为了排除源码控制元数据文件和目录，这一例外由userDefaultExclludes标记控制，其值默认为true。当userDefaultExcludes被开启的时候，所有如.svn/和CVS/的目录都会被排除在套件归档之外。第 12.5.3 节 “fileSets#######”显示了一个详细的默认排除模式的列表。

如果想要更多的对于文件集的控制，我们可以进一步显式的指定。例 12.7 “使用fileSet包含文件”展示了一个指定所有默认值的fileSet元素。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

例 12.7. 使用fileSet包含文件

```
<assembly>
  ...
  <fileSets>
    <fileSet>
      <directory>src/main/java</directory>
      <outputDirectory>src/main/java</outputDirectory>
      <includes>
        <include>**</include>
      </include>
      <useDefaultExcludes>true</useDefaultExcludes>
      <fileMode>0644</fileMode>
      <directoryMode>0755</directoryMode>
    </fileSet>
  </fileSets>
  ...
</assembly>
```

includes元素使用了一系列include元素，后者包含了路径模式。这些模式可能包含一些通配符，“**”表示匹配一个或者多个目录，“*”表示匹配文件名的任一部分，“？”表示匹配文件名中的任意单个字符。例 12.7 “使用fileSet包含文件”使用了一个fileMode元素来指定该文件集对于所有人可读，但是只有文件所有者可写。由于fileSet包含了目录，我们还可以指定directoryMode，它的使用方式和fileMode完全一样。一个目录的执行权限控制用户列出目录内容，我们这里想要确保目录对于所有人是可读且可执行的。和文件一样，只有目录所有者才拥有写的权限。

fileSet元素还提供了一些其它的配置选项。首先，它允许一个excludes子元素，其形式和includes完全一样。这些排除模式能让你从fileSet中排除特定的文件。包含模式优先于排除模式。此外，如果你想要将所包含文件中的表达式替换成属性值，你可以将filtering标记设置成true。表达式可以用${}标记（如org.sonatype.mavenbook）或者@@标记（这是标准的Ant表达式，如@project.groupId）来表示。你可以使用lineEnding元素来调整文件的换行字元；可用lineEnding值有：

keep
: 保持原始文件的换行字元。（这是默认值。）

unix
: Unix风格的字元

lf
: 仅仅一个换行符

dos
: MS-DOS风格的字元

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

crlf
　　回车后加一个换行符

最后，如果你想要确保所有文件匹配模式都被使用到，你可以使用userStrictFiltering元素，并将其值指定为true（默认值为false）。有些时候未使用的模式可能预示着某个中间输出目录缺失文件。使用值为true的useStrictFiltering后，当某个包含模式未被使用，Assembly插件就会失败。换句话说，如果你使用一个包含模式来包含构建中的某个文件，但是这个文件不存在，将userStrictFiltering设置成true会使得Maven构建失败。

12.5.3. fileSets#######

当你使用默认排除模式的时候，Maven Assembly插件不仅仅会忽略SVN和CVS信息。默认情况排除拥由Codehaus中plexus-utils[2]项目的DirectoryScanner[3]类定义。这个排除模式的数组被定义为一个static, final的String数组，其名称为DEFAULTEXCLUDES。该变量的内容如例 12.8 “Plexus Utils中模式排除模式的定义”所示。

[2] http://plexus.codehaus.org/plexus-utils/
[3] http://svn.codehaus.org/plexus/plexus-utils/trunk/src/main/java/org/codehaus/plexus/util/DirectoryScanner.java

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

例 12.8. Plexus Utils中模式排除模式的定义

```
public static final String[] DEFAULTEXCLUDES = {
     // Miscellaneous typical temporary files
     "**/*~",
     "**/#*#",
     "**/.#*",
     "**/%*%",
     "**/._*",

     // CVS
     "**/CVS",
     "**/CVS/**",
     "**/.cvsignore",

     // SCCS
     "**/SCCS",
     "**/SCCS/**",

     // Visual SourceSafe
     "**/vssver.scc",

     // Subversion
     "**/.svn",
     "**/.svn/**",

     // Arch
     "**/.arch-ids",
     "**/.arch-ids/**",

     //Bazaar
     "**/.bzr",
     "**/.bzr/**",

     //SurroundSCM
     "**/.MySCMServerInfo",

     // Mac
     "**/.DS_Store"
 };
```

这个默认的模式数组会排除来自于如GNU Emacs[4]的编辑器的临时文件，Mac中常见的临时文件，以及一些常见源码控制系统的元数据文件（虽然Visual SourceSafe得到的更

[4] http://www.gnu.org/software/emacs/

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

多的是恶名而非好的源码控制系统）。如果你需要覆盖这个默认的排除模式，你可以将setDefaultExcludes设置成false，然后在你自己的套件描述符中定义一组排除模式。

12.5.4. dependencySets 元素

套件中最常见的需求之一是包含项目的依赖。files和fileSets只是处理你项目中的文件，而依赖文件不存在于你的项目中。项目依赖的构件需要在构建过程中由Maven解析。依赖构件是抽象的，它们缺少明确的位置，它们使用一组Maven坐标来进行解析。相比较使用file和fileSets需要一个具体的资源路径，我们使用一组Maven坐标和依赖范围来包含或者排除依赖。

最简单的dependencySet是一个简单空元素：

```
<assembly>
  ...
  <dependencySets>
    <dependencySet/>
  </dependencySets>
  ...
</assembly>
```

上述的dependencySet会匹配你项目的所有运行时依赖（运行时范围隐式的包含编译范围依赖），并且它会将这些依赖添加到套件归档的根目录中。如果当前项目的主构件存在，它同时会复制该主构件至套件归档的根目录。

注意

等等？我之前认为dependencySet是用来包含我项目的依赖的，而不是项目的主构件。这一反直觉的副作用是Assembly插件版本2.1的一个bug，但是这个bug被广泛使用了，由于Maven强调向后兼容性，这一反直觉的，错误的行为就必须在2.1和2.2版中保持下来。但是，你可以控制这一行为，只要设置useProjectArtifact为false即可。

虽然没有任何配置的默认依赖集合十分有用，但该元素同时也支持很多配置选项，能让你定制其行为以适应你的环境。例如，第一件你想对依赖集合做的事情可能就是排除当前项目的构件，你会想设置userProjectArtifact为false（再次强调，由于历史原因，该配置的默认值为true）。这让你能够把项目输出和项目依赖分开管理。还有，你可能会将unpack标记为true（默认为false）来拆解依赖构件。当unpack被设置成true时，Assembly插件会组合所有匹配依赖的拆解内容至归档的根目录。

从这里你就可以看到，你有很多选择来控制依赖集合。下一节我们讨论如何定义依赖集合的输出位置，如何通过范围来包含和排除依赖。最后，我们会扩展依赖集的拆解功能，研究一些拆解依赖的高级选项。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

12.5.4.1. 自定义依赖输出目录

有两个配置选项可以用来协调定义依赖文件在套件归档中的位置：outputDirectory和outputFileNameMapping。你可能想要使用依赖构件自身的属性来定义其在套件中的位置。比如说你想要将依赖放到与其groupId对应的目录中。这时候，你可以使用dependencySet的outputDirectory元素，并提供如下的配置：

```
<assembly>
  ...
  <dependencySets>
    <dependencySet>
      <outputDirectory>org.sonatype.mavenbook</outputDirectory>
    </dependencySet>
  </dependencySets>
  ...
</assembly>
```

这会使得所有依赖被放到与其groupId对应的子目录中。

如果你想要更进一步的自定义，并移除所有依赖的版本号。你可以使用outputFileNameMapping来自定义每个输出文件的文件名，如：

```
<assembly>
  ...
  <dependencySets>
    <dependencySet>
      <outputDirectory>org.sonatype.mavenbook</outputDirectory>
      <outputFileNameMapping>
        ${module.artifactId}.${module.extension}
      </outputFileNameMapping>
    </dependencySet>
  </dependencySets>
  ...
</assembly>
```

这个例子中，依赖commons:commons-codec版本1.3最后会成为文件commons/commons-codec.jar。

12.5.4.2. 依赖输出位置的属性插值

正如在套件插值一小节所介绍的那样，outputDirectory和outputFileNameMapping不会和套件描述符的其它内容一样接受插值，这是因为它们的原始值必须使用额外的，构件特定的表达式解析器进行解释。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

对于这两个元素可用的构件表达式只有微小的差别。对两者来说，所有POM中及套件描述符其它部分中可用的${project.*}，${pom.*}，和${*}表达式，这里也能用。对于outputFileNameMapping元素来说，解析表达式的过程如下：

1. 如果表达式匹配模式${artifact.*}：

   a. 基于依赖的Artifact实例进行匹配（解析：groupId，artifactId，version，baseVersion，scope，classifier，和file.*）

   b. 基于依赖的ArtifactHander实例进行匹配（解析：expression）

   c. 基于和依赖Artifact相关的Project实例进行匹配（解析：主要是POM属性）

2. 如果表达式匹配模式${pom.*}或者${project.*}：

   a. 基于当前构建的项目实例（MavenProject)进行解析。

3. 如果表达式匹配模式${dashClassifier?}，而且Artifact实例包含一个非空的classfier，则解析成classifier前置一个破折号（-classfier）。否则，解析成一个空字符串。

4. 尝试基于当前构建的项目实例解析表达式。

5. 尝试基于当前项目的POM属性解析表达式。

6. 尝试基于系统属性解析表达式。

7. 尝试基于操作系统环境变量解析表达式。

outputDirectory也以差不多的方式进行插值，区别在于，对于outputDirectory没有可用的${artifact.*}信息，而只有特定构件的${project.*}实例信息。因此，上述罗列的的相关条目（上述处理过程列表中的1a，1b和3）就无效了。

我们怎么知道何时使用outputDirectory，何时使用outputFileNameMapping呢？当依赖被拆解的时候，只有outputDirectory会被用来计算输出路径。当依赖以完整的文件被管理时（不拆解），outputDirectory和outputFileNameMapping两者可以同时使用。在同时的时候，其结果等价于：

```
<archive-root-dir>/<outputDirectory>/<outputFileNameMapping>
```

在没有outputDirectory的时候，它便不被使用。当没有outputFileNameMapping的时候，其默认值为：

```
content-zh-0.6-SNAPSHOT${dashClassifier?}.${artifact.extension}
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

12.5.4.3. 通过范围包含或排除依赖

在第 9 章项目对象模型中，我们看到所有项目的依赖都有范围。依赖范围决定了依赖在构建过程的哪些阶段被使用。例如，test范围的依赖不会在编译项目主资源的时候被引入到classpath中；但当编译单元测试资源的时候，这些依赖就会被加入到classpath中。这是因为项目主资源的代码不该包含任何测试特有的代码，测试并不是一个项目的功能（它是项目构建过程的一个功能）。类似的，provided范围的依赖会被认为是已经在最终的部署环境中存在。但是，如果一个项目依赖于某个特定的provided依赖，那么它还是需要这个依赖以进行编译。因此，provided范围的依赖存在于编译classpath中，但是不会和项目构件或套件一起被打包发布。

同样在第 9 章项目对象模型中，可以回忆到，一些依赖范围暗指了其它依赖范围。例如，runtime依赖范围暗指了compile范围，代码运行的时候需要所有编译时的依赖（除去那些provided范围的依赖）。各种各样的依赖范围之前有很多复杂的关系，这些关系控制一个直接依赖的范围如何影响间接依赖的范围。在Maven套件描述符中，我们可以使用依赖范围将不同的设置应用到不同的依赖组中。

例如，如果我们打算在web应用中包裹Jetty[5]以创建一个完全自包含的应用，我们会需要包含所有jetty目录结构中的所有provided范围依赖。这才能确保那些provided范围的依赖存在于实际的运行环境着哦国内。非provided的，运行时的依赖仍然位于WEB-INF/lib目录中，因此这两组依赖需要分别处理。这些依赖组可能看起来很类似，如下列XML：

例 12.9. 使用范围来定义依赖组

```xml
<assembly>
  ...
  <dependencySets>
    <dependencySet>
      <scope>provided</scope>
      <outputDirectory>lib/content-zh</outputDirectory>
    </dependencySet>
    <dependencySet>
      <scope>runtime</scope>
      <outputDirectory>
        webapps/${webContextName}/WEB-INF/lib
      </outputDirectory>
    </dependencySet>
  </dependencySets>
  ...
</assembly>
```

[5] http://www.mortbay.org/jetty-6/

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

provided范围的依赖会添加到套件根目录下的lib/目录中，该目录是一个会被包含到Jetty全局运行classpath中的类库目录。我们使用根据项目artifactId命名的子目录以方便跟踪某个类库的出处。运行时依赖被添加到web应用的WEB-INF/lib路径下，这位于标准jetty的webapps/下的一个子目录中，该子目录使用自定义的POM属性webContextName进行设置。在这个例子中我们完成的事情是，将应用程序特定的依赖与那些位于Server容器全局classpath中的依赖隔离。

然而，简单的根据范围来分隔依赖可能还不够，尤其是在web应用中。我们可以想象到某个或者多个运行时依赖可能实际上是供web应用使用的标准化的，非二进制的资源包裹。例如，考虑有一组web应用程序重用了一组Javascript, CSS, SWF, 和图片资源。为了是这些资源更容易标准化，通常的做法是将它们包裹成一个归档文件并部署到Maven仓库中。这样，它们就可以通过标准的Maven依赖被引用——可能依赖的类型是zip——而这样的依赖通常是runtime范围的。但是，别忘了，它们是资源文件，不是应用程序代码本身的二进制依赖；因此，盲目的将这些文件放到WEB-INF/lib目录中是错误的。我们应该把这些资源归档和其它runtime依赖区别对待，将它们放到拆解后放到web应用的文档根目录中。为了实现这种分离，我们需要使用可以应用到特定依赖坐标上的包含和排除模式。

换句话说，假如你有三个或者四个web应用，它们重用了同样的资源。你想要创建一个套件，将provided依赖放到lib/，将runtime依赖放到webapps/<contextName>/WEB-INF/lib，将某个特定的runtime依赖拆解并放到web应用的文档根目录中。你能完成这个任务，因为Assembly允许你为某个dependencySet元素定义多个包含和排除模式。阅读下一节将帮助你了解更多相关内容。

12.5.4.4. 微调：依赖包含和排除

一个资源依赖可能只是简单的某个项目的一组资源（CSS, Javascript, 和图片），该项目为此创建了一个ZIP归档格式的套件。基于我们web应用的特性，我们也许可以简单的通过依赖的打包格式来区分资源依赖和其它字节码依赖。大部分web应用以jar的形式依赖其它项目，因此我们可以简单的决定说有zip依赖为资源依赖。又或者，可能某些时候资源以jar的格式存储，但我们可以使用一个分类器，如resources。不管哪种情形，我们都可以为这些资源依赖使用包含模式，同时为其它字节码依赖使用不同的逻辑。我们会使用dependencySet的includes和excludes子元素来指定这些微调模式。

includes和excludes都包含一子元素，相应的为include和exclude子元素。每个include或者exclude元素包含一个字符串值，该值可以包含通配符。每个字符串值都可以通过很多不同方式匹配依赖。总的来说，这里支持三种定义符匹配格式：

`groupId:artifactId` - 无版本id
: 仅仅使用groupId和artifactId来匹配依赖。

`groupId:artifactId:type[:classifier]` - 冲突id
: 该模式能让你指定一个更广的坐标范围，以创建更具体的包含/排除模式。

groupId:artifactId:type[:classifier]:version - 完整构件定义符
如果你需要十分十分的具体，你可以指定所有坐标。

所有这三种模式都支持通配符‘*’，它可以匹配定义符的任何一个子段，但不局限于一个单独的子段（位于‘:’中间的部分）。同时，注意上述的分类器部分是可选的，如果需要匹配的依赖没有分类器，就不需要在模式中指定分类器子段。

在前面讲述的例子中，关键的区别在于构件类型zip，并且没有任何依赖拥有分类器，下列的模式会匹配所有类型为zip的资源依赖：

```
*:zip
```

上述的模式使用了第二种依赖定义符：依赖的冲突id。既然我们拥有了区分资源依赖和其它字节码依赖的模式，我们可以修改依赖集合来区分处理资源归档：

例 12.10. 在dependencySets中使用依赖排除和包含

```
<assembly>
  ...
  <dependencySets>
    <dependencySet>
      <scope>provided</scope>
      <outputDirectory>lib/content-zh</outputDirectory>
    </dependencySet>
    <dependencySet>
      <scope>runtime</scope>
      <outputDirectory>
        webapps/${webContextName}/WEB-INF/lib
      </outputDirectory>
      <excludes>
        <exclude>*:zip</exclude>
      </excludes>
    </dependencySet>
    <dependencySet>
      <scope>runtime</scope>
      <outputDirectory>
        webapps/${webContextName}/resources
      </outputDirectory>
      <includes>
        <include>*:zip</include>
      </includes>
      <unpack>true</unpack>
    </dependencySet>
  </dependencySets>
  ...
</assembly>
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

在例 12.10 “在dependencySets中使用依赖排除和包含”中，前一个例子中的运行时依赖集合已经更新，现在它排除了资源依赖。我们的字节码依赖（非zip依赖）需要被添加到web应用的WEB-INF/lib目录中。而资源依赖现在有其自身的依赖集合，它被配置成包含那些zip依赖，并输出至web应用的资源目录。例子中最后一个dependencySet中的includes元素和前一个dependencySet中的excludes元素起到的效果完全相反，因此使用同样的定义符模式，资源依赖就被准确包含了。最后一个dependencySet元素指向了共享资源依赖，并会拆解资源依赖至web应用的文档根目录中。

例 12.10 “在dependencySets中使用依赖排除和包含”的前提是所有共享资源依赖的类型和其它依赖类型不一样。那么如果共享资源依赖和其它依赖的类型一致呢？你如何区分这些依赖？在这种情况下，假如所有的共享资源依赖都被打包成JAR，并且其分类器是resources，你就可以更改定义符模式来匹配这些依赖：

```
*:jar:resources
```

这里不再匹配那些类型为zip且没有分类器的构件，我们匹配所有类型为jar且分类器为resources的构件。

和fileSets元素一样，dependencySets元素也支持useStrictFiltering标记。开启这个标记的时候，当任何一个指定的模式没有匹配任何依赖的时候，assembly插件运行就会失败，因此构建也会失败。有时候这是个很有用的保障，可以用来保证你项目的依赖和套件描述符是同步的，且按照你的意愿相互作用。默认情况下，为了向前兼容，该标记是关闭的。

12.5.4.5. 传递性依赖，项目构件，项目附属构件

dependencySet元素支持两种更通用的机制来微调一小组匹配构件：传递性依赖选项，以及操作项目构件的选项。这两种特性都是为了支持遗留配置，这些配置所理解的“依赖”条件更为宽松。举个典型的例子，让我们考虑项目本身的构件。一般来说，这不会被算作是一个依赖；然而老版本的Assembly插件把项目本身的构件也算作依赖集。为了给这种“特性”提供向前兼容性，Assembly插件的2.2版本（目前是2.2-beta-2）支持一个名为useProjectArtifact的标记，其默认值为true。因此默认情况依赖集合在计算包含和排除的时候会考虑项目本身构件。如果你要单独处理项目构件，就需要将该标记置为false。

> **提示**
>
> 本书的作者推荐你一直将useProjectArtifact置为false。

自然的，作为项目构件的扩展，项目的附属构件也可以通过使用useProjectAttachments标记（默认值为false）在dependencySet中进行管理。开启这个标记后，指定分类器和类型的模式就会匹配那些“附属到”项目主构件的构件；就是说，它们共享基本的groupId/artifactId/version定义符，但是他们的type和

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

classifier和主构件不同。当我们需要在套件中包含JavaDoc或者源码文件的时候，就可以使用这一特性。

除了处理项目自身的构件，我们还能够微调使用两个传递性依赖解析标记来微调依赖集合。首先，名为useTransitiveDependencies（默认为true）指定依赖集合是否计算传递性依赖。举个实际的例子，如果你的POM依赖于另外一个套件，这个套件（很可能）会有一个分类器以区分其主构件并将自身作为一个附属构件。然而，Maven在解析依赖的时候，就算是解析套件构件，其主构件的传递性依赖信息还是会被解析。如果套件已经将项目依赖包裹，使用传递性依赖解析就会造成这些依赖的重复。为了避免这一情况，我们重要简单的将useTransitiveDependencies置为false。

另外一个传递性依赖解析标记就更微妙了。它叫做useTransitiveFiltering，其默认值为false。要了解这个标记是做什么的，我们首先需要理解在解析过程中哪些信息是对所有任何构件可用的。当一个构件是依赖的依赖的时候（就是说，需要在你项目本身的POM再往下至少一层），它有一个被Maven称作“依赖踪迹“的东西，依赖踪迹维护一个字符串序列，该序列包含一列完整的构件定义符（groupId:artifactId:type:[classifier:]version），所有你项目的POM和拥有该依赖踪迹的构件的定义符都在其中。如果你还记得依赖集合模式匹配中可用的三种构件定义符，你会注意到依赖踪迹的每一项——完整构件定义符——对应了第三种类型。在useTransitiveFiltering为true的时候，依赖踪迹中的条目就会像主构件一样被包含或排除。

如果你考虑使用传递性依赖过滤（transitive filtering），要小心！某个构件可能会在传递依赖图中被包含多次，但是自Maven 2.0.9之后，只有第一个包含的踪迹会被认作该类型的匹配。当聚集你项目的依赖的时候，这可能会造成微妙的问题。

> 警告
>
> 大部分套件实际上不需要对于依赖集合这么深层次的控制；当你真要这么做的时候，仔细考虑，很可能没有这个必要。

12.5.4.6. 高级拆解选项

正如我们之前讨论的，一些项目依赖需要先被拆解以帮助创建可用的套件归档。在前面的例子中，是否要拆解很明显。但是，没有考虑哪些部分需要拆解，或者，更重要的，哪些部分不要拆解。为了更好的控制拆解过程，我们可以配置dependencySet的unpackOptions元素。使用该元素，我们就可以选择拆解的时候包含或者排除那些文件模式，并且还可以配置那些被包含的文件是否需要被过滤以使用当前POM信息解析表达式。事实上，关于拆解依赖集合可用的选项，与包含项目目录结构使用的fileSets可用的选项十分类似。

我们继续前面的web应用样例，假设一些资源依赖包裹了分发许可证文件。而在我们的web应用中，我们希望在自己套件的NOTICES文件中集中处理许可证信息，因此我们不希

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

望引入该资源依赖的许可证文件。要排除这个文件，我们只需要在依赖集合配置中添加拆解选项如下：

例 12.11. 在依赖拆解的时候排除文件

```
<asembly>
  ...
  <dependencySets>
    <dependencySet>
      <scope>runtime</scope>
      <outputDirectory>
        webapps/${webContextName}/resources
      </outputDirectory>
      <includes>
        <include>*:zip</include>
      </includes>
      <unpack>true</unpack>
      <unpackOptions>
        <excludes>
          <exclude>**/LICENSE*</exclude>
        </excludes>
      </unpackOptions>
    </dependencySet>
  </dependencySets>
  ...
</assembly>
```

注意这里我们使用的exlude元素和前面fileSet中使用的exlude元素十分类似。这里，我们拦截资源依赖目录中所有以LICENSE名字开头的文件。你可以将拆解选项看成是一个应用到每个匹配依赖上的轻量级fileSet配置。换句话说，它是一个应用到拆解依赖上的fileSet配置。和为资源依赖指定文件排除模式一样，我们也可以使用includes元素严格指定一组被包含的文件。在fileSets中用来处理包含和排除的同样的代码在unpackOptions这里是被完全重用的。

除了包含和排除，拆解选项同样提供了filtering标记，其默认的值为false。同样，我们在fileSets中讨论该选项。对于两者来说，使用Maven语法的${property}和使用Ant语法的@property@都被支持。然而，对于依赖集合来说，过滤是一个非常有用的特性，因为这能让你建立标准化的，版本化的资源模板，并且每个套件在包含该模板的时候都可以对其进行自定义。一旦你掌握了如何使用过滤的，拆解的，包含共享资源的依赖，你就可以能抽象出那些重复的资源文件，生成一个通用的资源项目。

12.5.4.7. 依赖集合小结

最后，值得一提的是，依赖集合和文件集合一样支持同样的fileMode和directoryMode配置选项，但是你应该记住directoryMode只有在依赖被拆解的时候被使用。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

12.5.5. moduleSets 元素

多模块构建通常会使用相互关联POM中的parent和modules元素来凝聚在一起。典型的来说，父POM在modules元素中指名其子模块，一般情况这就会使得子POM被包含到父项目的构建过程中。这种关系的具体定义对Assembly插件在构建过程中起的作用会造成很大影响，之后我们会详细讨论这一点。目前，只要求你在我们讨论moduleSets元素的时候记住父子关系。

很多项目被凝聚在一个多模块构建中是因为它们是一个更大系统的一部分。这些项目被设计成一起使用，在一个大型构建中的一个单独模块本身没有太大的实用价值。从某方面来说，项目构建的结构和我们期望该项目（以及它的模块）被使用的方式相关。如果从用户的角度考虑这个项目，那么可以认为构建的最终理想目标是提供一个单独的，可分发的，用户无须花太多安装的文件。由于Maven多模块项目基本上都遵循一个上下结构，其依赖信息，插件配置，还有其它信息自从父到子被继承，那么很自然的，我们会想到将所有模块凝聚成一个单独分发包的这样一个工作，最终会落到顶层项目的头上。这就是为什么会有moduleSet元素的原因。

模块集合能让属于项目结构中每个模块的资源被包含到最终的套件归档中。就像你可以使用fileSet和dependencySet选择包含一组文件一样，在一个多模块构建中，你可以使用moduleSet指定包含某些模块的文件和资源。通过激活两个基本类型的，模块特定的包含元素就可以实现这一点：file-based，和artifacts-based。在我们深入了解基于文件包含和基于构件包含之前，我们先简单讨论一下选择处理哪些模块。

12.5.5.1. 模块选择

到目前为止，你应该已经熟悉includes/excludes模式了，它们贯穿了整个套件描述符，用来过滤文件和依赖。当你在套件描述符中指向模块的时候，你就需要使用includes/excludes模式来定义规则，这些规则会被应用到一组模块上。moduleSet中的includes和excludes元素不同的地方在于它们不允许使用通配符模式。（到2.2-beta-2版本为止，还没看到这个特性有太大的需要，因此仍然未实现。）每个Include或者exclude的值都只是简单的某个模块的groupId和artifactId，通过冒号分隔，如：

```
groupId:artifactId
```

除了includes和excludes，moduleSet还支持一种额外的选择工具：includeSubModules标记（默认值为true）。多模块构建中的父子关系并不会严格的限制在两层。事实上，你可以包含任意层的模块。一个当前项目模块包含的模块被认为是一个子模块。某些情况，你想要单独的处理构建中的每一个模块（包括子模块）。例如，通常处理基于构件的贡献的最简单的方法是在这些模块中处理，为此，你只要简单的让includeSubModules为默认值true。

当你试图从每个模块的目录结构中包含文件的时候，你可能会希望一次整个的处理所有模块的目录。如果你的项目目录结构和POM中的父子结构一致，这种方式就能让模式如

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

**/src/main/java不仅仅应用到当前模块的目录中，同时也应用到子模块的目录中。这种情况下你不想直接处理子模块（它们会以当前项目子目录的形式被处理），你应该将includeSubModules设置成false。

在我们决定了选择哪些模块被处理之后，我们就可以选择每个模块包含的内容了。正如前面提到的，我们可以选择包含模块项目的文件或者构件。

12.5.5.2. 源码选择

假设你想要在套件中包含所有模块的源码，但是你希望排除某个特殊的模块。比如说可能你有一个名为secret-sauce的项目，它包含了一些你不想随着项目分发的秘密且敏感的代码。要做到这一点，最简单的方式是使用moduleSet，它在${module.basedir.name}中包含每个项目的目录，并且将secret-sauce排除在套件之外。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

例　12.12.　使用moduleSet包含和排除模块

```
<assembly>
  ...
  <moduleSets>
    <moduleSet>
      <includeSubModules>false</includeSubModules>
      <excludes>
        <exclude>
          com.mycompany.application:secret-sauce
        </exclude>
      </excludes>
      <sources>
        <outputDirectoryMapping>
          ${module.basedir.name}
        </outputDirectoryMapping>
        <excludeSubModuleDirectories>
          false
        </excludeSubModuleDirectories>
        <fileSets>
          <fileSet>
            <directory>/</directory>
            <excludes>
              <exclude>**/target</exclude>
            </excludes>
          </fileSet>
        </fileSets>
      </sources>
    </moduleSet>
  </moduleSets>
  ...
</assembly>
```

在例　12.12 “使用moduleSet包含和排除模块”中，由于我们是在处理每个模块的源码，因此只处理当前项目模块更简单，我们可以使用文件集合的文件路径通配符来处理子模块。这里我们将includeSubModules元素设为false，因此就不用担心子模块出现在套件的根目录中。exclude元素会负责排除secret-sauce模块。我们就不会包含这个秘密模块的源码了。

一般来说，包含的模块源码会位于套件中一个名为模块artifactId的子目录中。然而，因为Maven允许模块不位于根据其artifact命名的目录中，因此通常最好使用表达式${module.basedir.name}来保证模块目录的实际名称（${module.basedir.name}与MavenProject.getBasedir().getName()的值一致）。一定要记住模块并不是一定以子目录的形式存在。如果你的项目有一个十分奇怪的目录结构，你可能就需要凭借特殊的moduleSet声明来包含特定的项目，以及解释你项目的特异性。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

警告

尽量降低你项目的特异性，虽然Maven很灵活，但如果你发现你配置太多，就很可能有一种更简单的方式。

继续讨论例 12.12 “使用moduleSet包含和排除模块”，由于我们没有在这个模块集合中显式的处理子模块，我们需要确保子模块的目录内容没有被排除在外。通过设置excludeSubModuleDirectories为true，我们就可以在子模块的目录中应用相同的文件模式。最后在例 12.12 “使用moduleSet包含和排除模块”中，我们对模块集合构建过程中的任何输出都没有兴趣，因此我们从所有模块中排除target/目录。

还有一点值得一提的是，除了支持嵌套fileSets以外，sources元素本身直接支持所有类fileSet的元素。这些配置元素用来提供对之前Aseembly插件版本（版本2.1及之前）的向前兼容性，之前的版本在没有创建一个单独的模块集合声明的情况下，不支持同一模块的多个文件集合。这种用法是被废弃的，不应该被使用。

12.5.5.3. moduleSets#outputDirectoryMapping###

在第 12.5.4.1 节 “自定义依赖输出目录”中，我们使用元素outputDirectoryMapping来更改包含每个模块源码的目录名称。解析该表达式元素的方式和依赖集合中解析outputFileNameMapping元素的方式完全一致（见第 12.5.4 节 “dependencySets 元素”中该算法的解释）。

在例 12.12 “使用moduleSet包含和排除模块”中，我们使用了表达式${module.basedir.name}。你可能会注意到该表达式的根元素，module，并没有在依赖集合的匹配解析算法中列出；这个根元素moduleSets配置特有的。它其实和outputFileNameMapping元素中使用${artifacts.*}引用一样，只是这里应用到了模块的MavenProject， Artifact， 和ArtifactHandler上，而非依赖构件。

12.5.5.4. 字节码选择

源码选择主要关注以源码的形式包含一个模块，而字节码选择就主要关注包含模块的构建输出，或者说构件。虽然这一部分运行就效果就像声明一个依赖集合并应用到每个模块上，但这里还有一些额外的特性只对模块构件有效，这些元素值得讨论：attachmentClassifier和includeDependencies。此外，biniaries元素包含了一些与dependencySet元素类似的选项，它们关系到模块构建本身的处理。它们是：unpack，ouputFileNameMapping， outputDirectory， directoryMode， 以及fileMode。最后，模块字节码可以包含一个dependencySets元素，用来指定每个模块的依赖应该如何被包含到套件归档中。首先，让我们看一下这里提到的选项如何被用来管理模块本身的构件。

假设你想要在套件中包含每个模块的javadoc jar。这时，我们不考虑包含模块依赖；我们只要javadoc jar。然而，由于这个特殊的jar是项目主构件的一个附属构件，我们需要指定它的分类器以获取该jar。简单起见，我们不会涉及到拆解该模块的

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

jajvadoc jar，拆解的配置和我们之前配置依赖集合完全一样。最后的模块集合看起来会像例 12.13 “在套件中包含模块的JavaDoc”。

例 12.13. 在套件中包含模块的JavaDoc

```
<assembly>
  ...
  <moduleSets>
    <moduleSet>
      <binaries>
        <attachmentClassifier>javadoc</attachmentClassifier>
        <includeDependencies>false</includeDependencies>
        <outputDirectory>apidoc-jars</outputDirectory>
      </binaries>
    </moduleSet>
  </moduleSets>
  ...
</assembly>
```

在例 12.13 “在套件中包含模块的JavaDoc”中，我们没有显式的设置includeSubModules标记，它默认为true。我们当然想要使用该模块集合来处理所有的模块，包括子模块，因为我们没有使用任何可以匹配子模块目录结构的文件模式。attachmentClassifier会抓取每个被处理模块的分类器为classifier的附属构件。而includeDependencies元素告诉Assembly插件，我们对模块的依赖没有兴趣。最后，outputDirectory元素告诉Assembly插件将所有javadoc jar放到套件跟目录下名为apidoc-jars的目录中。

虽然在本例中我们没有做什么特别复杂的配置，但重点需要理解的是，source元素中关于outputDirectoryMapping子元素讨论的表达式解析算法这里同样适用。也就是说，任何dependencySet的outputFileNameMapping配置中可用的`${artifact.*}`，在这里都可以通过`${module.*}`的形式使用。它们起到的效果是一样的。

最后，让我们看一个例子，这里我们想要处理模块的构件以及运行时依赖。在这种情况下，我们需要根据模块的artifactId和version将每个模块的构件集合分离到单独的目录结构中。最终的模块结合惊人得简单，就像例 12.14 “在套件中包含模块构件和依赖”：

例 12.14. 在套件中包含模块构件和依赖

```
<assembly>
  ...
  <moduleSets>
    <moduleSet>
      <binaries>
        <outputDirectory>
          ${module.artifactId}-${module.version}
        </outputDirectory>
        <dependencySets>
          <dependencySet/>
        </dependencySets>
      </binaries>
    </moduleSet>
  </moduleSets>
  ...
</assembly>
```

在例 12.14 “在套件中包含模块构件和依赖”中，我们使用了空的dependencySet元素，这时因为默认情况下，没有任何配置它就会包含所有运行时依赖。这里我们设置了字节码集合的outputDirectory，所有依赖以及模块主构件都会被包含到这个目录中，因此，我们设置不需要再去配置依赖集合。

很大程度上，模块字节码直接明了。两个部分——主要部分，关注处理模块构件本身，和依赖集合，关注模块依赖——它们的配置选项都和依赖集合的配置类似。当然，binaries元素同时还提供了一些选项用来控制是否包含依赖，以及你想要使用那个主项目构件。

如同sources元素一样，binaries元素包含了一些目的仅为向前兼容性的元素，它们应该被弃用。这包括了includes和excludes子元素。

12.5.5.5. **moduleSets**，父POM，和**binaries**元素

最后，我们用一个警告来结束模块处理的讨论。Maven内部设计中关于父-模块关系的部分与模块集合binaries元素的运行有着微妙的相互影响。当一个POM声明一个父项目，那么父项目就必须在当前项目被构建之前以某种方式解析。如果父项目在Maven仓库中，这没什么问题。然而，从Maven 2.0.9起，如果父项目是同一构建下的上一层POM，这就会出现问题，尤其是在父POM期望使用子项目的字节码构建套件的时候。

Maven 2.0.9根据依赖关系来安排多模块项目构建的顺序，某个模块的依赖总在该模块之前被构建。问题在于，父项目也被看成是一个依赖，也就是说，父项目的构建必须在子项目构建之前就完成。如果父项目构件的过程包括创建一个套件，而这个套件使用了模块字节码，而这些字节码还不存在，也就不能被包含，那么Assembly插件就会失

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

败。这是一个复杂且微妙的问题，严重影响了套件描述符模块字节码小节的用途。事实上，已经有一个Assembly插件的bug记录来跟踪这一问题：http://jira.codehaus.org/browse/MASSEMBLY-97。希望未来版本的Maven能够找到方法来修复这一功能，因为很多时候父项目必须首先构建这一需求并不是完全必要。

12.5.6. Repositories元素

套件描述符中的repositories元素是可以说是一个舶来品，因为除了Maven自己很少有应用能够利用Maven仓库的目录结构。此外，repositories元素中的很多功能与dependencySets元素十分类似，因此我们这里不花太多时间在这上面。大部分情况下，理解依赖集合的用户都能很轻松的在套件描述符中编写正确的repositories元素。我们就不详细介绍repositories元素了；也不会再建立一个用例再一步步解释。但是，如果你发现自己需要使用repositories元素，我还是警告你小心。

说过这些之后，这里值得一提的是两个repositories元素特有的特征。第一个是includeMetadata标记。当它为true的时候，套件就会包含仓库元数据，如与-SNAPSHOT虚拟版本对应的一列真实版本，默认情况下，该标记为false。目前，当这个标记为true的时候，只有从Maven中央仓库下载的元数据会被包含进来。

第二个特性名为groupVersionAlignments。同样，该元素包含一列单独的groupVersionAlignment子元素，该子元素的目的是规范某个特定groupId下包含的所有构件使用一个单独的版本。每个groupVersionAlignment元素包含两个必须的子元素——id和version——还包含可选的excludes元素，该excludes元素可以包含一列artifactId的字符串值，使其被排除在外。不幸的是，这种版本的校正不会修改仓库中的POM，不管是被校正的构件还是依赖于它们的构件，其POM都不会被修改，因此难以想象实际用到这种校正的应用会是什么样子。

总的来说，当你添加repositories元素的时候，同样遵循使用依赖集合的原则。虽然repositories元素支持上述额外的特性，但这主要还是为了提供向后兼容性，因此以后的版本中可能会被废弃。

12.5.7. 管理套件的根目录

到现在为止我们仔细了解的套件描述符的主体部分，我们可以用相对轻量级的内容来结束关于描述符内容相关的叙述：根目录命名和站点目录处理。

有人可能认为这只是形式考虑，但是拥有对于套件根目录命名的控制，或者是否需要根目录，都通常很重要。幸运的是，套件描述符根元素下的两个配置选项使得管理归档根目录变得简单，它们是：includeBaseDirectory和baseDirectory。如果套件是一个可运行jar文件，你可能就完全不想要根目录。为了忽略根目录，只要简单的将includeBaseDirectory设置为false（默认为true）。这样，生成的归档在打开的时候，就可能在拆解目标目录创建多个目录。如果一个归档在使用之前需要拆解，这通常是一种不好的方式，最好是一个归档能够照原样使用。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

在一些其它情况，你可能想要保证归档根目录的名称与POM的版本或者其它信息一致。默认情况下，baseDirectory的值就等于content-zh-0.6-SNAPSHOT。但是，我们可以很简单的更改这个元素的值，可以是一个字符串字面量，或者是一个根据当前POM进行插值的表达式，比如org.sonatype.mavenbook-content-zh。对你的文档团队来说，这是个好消息！（我们都拥有这样的团队，是吧？）

另外一个可用的配置选项是includeSiteDirectory，其默认值是false。如果你项目的构建同时使用了site生命周期的Site插件目标创建了web站点文档，你就可以通过将该选项设置成true来包含这些内容。不过，这个特性有一些限制，只有当前POM的文档部分会被包含（默认情况是target/site），而所有子模块的站点目录不在考虑之中。如果你想要用，就用吧，不过一个适当配置的fileSet或者moduleSet可以完成同样的任务，可能还会更好。其实这里又是一个Assembly插件考虑向前兼容性的例子，你最好还是使用fileSet或者moduleSet元素，而非将includeSiteDirectory设置成true。

12.5.8. **componentDescriptors**和**containerDescriptorHandlers**

为了丰满我们关于套件描述符的讨论，我们还应该简单
看一下另外两个元素：containerDescriptorHandlers和componentDescriptors。containerDescriptorHandlers元素指向一个自定义的组件，你可以用该组件来扩展Assembly插件的功能。具体来说，这些自定义的组件能让你定义及处理一些特殊的文件，它们可能需要被从各个成分（constituent）归并到一起以创建你的套件。举个很好的例子，可能有一个自定义容器描述符处理器，用来从一个成分war或者war片段文件归并web.xml到你的套件中，这样你就可以在最终的套件中保持一个单独的web应用描述符，并将其看成一个war文件。

componentDescriptors元素允许你引用外部的套件描述符片段，然后将其包含到当前的描述符中。组件引用可以是如下任何一种：

1. 相对文件路径： src/main/assembly/component.xml

2. 构件引用： groupId:artifactId:version[:type[:classifier]]

3. Classpath资源： /assemblies/component.xml

4. URL： http://www.sonatype.com/component.xml

顺便说一下，在解析组件描述符的时候，Assembly插件严格的按照上面的顺序采取不同的策略。第一个成功的策略会被使用。

组件描述符可以包含大多数组件描述符中可用的关于内容的元素，但是moduleSets除外，这时因为该元素被认为是每个项目特定的，不该被重用。组件描述符中还可以包含的元素是containerDescriptorHandlers，这个之前已经简要过。组件描述符不能包含格式，套件ID，或者其它任何与套件归档基础目录相关的配置，所有这些配置都是特定

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

套件描述符唯一的。虽然共享格式元素是有道理的，但在2.2-beta-2版本的Assembly插件中，还没有实现。

12.6. 最佳实践

Assembly插件提供了足够的灵活性，以很多不同的方式处理许多问题。如果你的项目有特定的需求，那么你完全可以使用本章叙述的方法来实现几乎任何的套件结构。本节，我们讨论讨论一些最佳实践，如果你遵循它们，你就能更高效的使用Assembly插件，同时也能减少不必要的痛苦。

12.6.1. 标准的，可重用的套件描述符

到目前位置，我们主要讨论了一些针对某个特定套件的一次性解决方案。但是如果你有几十个项目需要某个特定类型的套件怎么办？简言之，我们如何才能重用我们为某个套件所花的精力，并将其使用到多个项目中去，而不是复制粘贴套件描述符？

最简单的答案是从套件描述符中创建一个标准化的，版本化的构件，并将其部署。一旦完成了这一步，你就可以在你项目的POM中指定Assembly插件的plugin元素以插件级别依赖的形式去使用套件描述符构件，这会让Maven解析并包含套件描述符至插件的classpath中。这时，你就可以通过配置Assembly插件的descriptorRefs元素来使用这个套件描述符。请看如下的套件描述符：

```
<assembly>
  <id>war-fragment</id>
  <formats>
    <format>zip</format>
  </formats>
 <includeBaseDirectory>false</includeBaseDirectory>
  <dependencySets>
    <dependencySet>
      <outputDirectory>WEB-INF/lib</outputDirectory>
    </dependencySet>
  </dependencySets>
  <fileSets>
    <fileSet>
      <directory>src/main/webapp</directory>
      <outputDirectory>/</outputDirectory>
      <excludes>
        <exclude>**/web.xml</exclude>
      </excludes>
    </fileSet>
  </fileSets>
</assembly>
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

在你的项目中使用它，所有项目的内容就会被打包在一起，然后可以直接解压添加到现存web应用中（为了添加一个扩展特性）。然而，如果你的团队构建了很多个web片段项目，那么你就会想要重用该描述符，而非复制它。为了将该描述符以构件的形式部署，我们就要将其放到一个项目中，位于src/main/resources/assemblies目录。

这个套件描述符构件的项目结构看起来如下：

```
|-- pom.xml
`-- src
    `-- main
        `-- resources
            `-- assemblies
                `-- web-fragment.xml
```

注意web-fragment.xml文件的位置。默认情况下，Maven会将整个src/main/resources目录打包到最终的jar中，这里，如果没有额外的配置，套件描述符也就会被打包。同时注意assemblies/这一路径前缀，Assembly插件需要所有插件classpath中的描述符拥有这一前缀。将我们的描述符放到恰当的相对路径十分重要，这样Assembly才能够在运行的时候识别它们。

记住，现在这个项目已经和你实际的web-fragment项目分离了；套件描述符拥有了自己的构件，自己的版本，甚至有可能的话，自己的发布周期。一旦你使用Maven install了这一新的项目，你就能够在web-fragment项目中使用它。如果还不清楚，检查构建过程，看起来应该如下：

```
$ mvn install
(...)
[INFO] [install:install]
[INFO] Installing (...)/web-fragment-descriptor/target/web-fragment-descriptor-1.0-
       to /Users/~/.m2/repository/org/sonatype/mavenbook/assemblies/\
          web-fragment-descriptor/1.0-SNAPSHOT/web-fragment-descriptor-1.0-SNAPSHOT
[INFO] ------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------
[INFO] Total time: 5 seconds
(...)
```

由于web-fragment-descriptor项目没有源码，得到的jar除了包含web-fragment套件描述符什么都没有。现在，让我们使用这个新的描述符构件：

```
<project>
  (...)
  <artifactId>my-web-fragment</artifactId>
  (...)
  <build>
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛


```
    <plugins>
      <plugin>
        <artifactId>maven-assembly-plugin</artifactId>
        <version>2.2-beta-2</version>
        <dependencies>
          <dependency>
            <groupId>org.sonatype.mavenbook.assemblies</groupId>
            <artifactId>web-fragment-descriptor</artifactId>
            <version>1.0-SNAPSHOT</version>
          </dependency>
        </dependencies>
        <executions>
          <execution>
            <id>assemble</id>
            <phase>package</phase>
            <goals>
              <goal>single</goal>
            </goals>
            <configuration>
              <descriptorRefs>
                <descriptorRef>web-fragment</descriptorRef>
              </descriptorRefs>
            </configuration>
          </execution>
        </executions>
      </plugin>
      (...)
    </plugins>
  </build>
  (...)
</project>
```

在这个Assembly插件配置中，有两个特殊的地方：

- 这里必须包含一个插件级别依赖声明，只有依赖了web-fragment-descriptor构件之后我们才能通过插件classpath访问套件描述符。
- 由于我们使用classpath引用，而非本地项目目录结构的文件，我们就必须使用descriptorRefs元素，而非descriptor。还有要注意的是，虽然套件描述符实际上是位于插件classpath的assemblies/web-fragment.xml位置，但我们这里引用的时候没有写明assemblies/前缀。这是因为Assembly插件假设所有内置的套件描述符都永远位于classpath下的该前缀位置下。

现在，你可以随意的在任何多个项目中重用这部分POM配置，并能确保它们所有的web-fragment套件是相似的。可能你需要调整套件格式——可能是为了包含其它资源，或者

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

为了微调依赖和文件集合——你可以提升套件描述符项目的版本，然后再次发布。引用该套件描述符的POM就可以使用新的版本来生效。

关于套件描述符重用最后一点：你可能不仅仅想要将描述符发布成一个构件，你还希望共享插件配置。这很简单；只要将上述的插件配置放到父POM的pluginManagement元素中，然后在子模块的POM中如下引用这段插件配置：

```
(...)
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-assembly-plugin</artifactId>
      </plugin>
(...)
```

如果你已经将前面的插件配置放到了父POM中的pluginManagement中，那么所有继承该POM的项目就可以使用如上最少的配置，正确的引用套件描述符，实现我们自定义的高级的套件格式。

12.6.2. 分发（聚合）套件

正如之前提到的，Assembly插件提供很多种方式来创建不同的归档格式。分发包是一个典型的例子，因为它们通常结合了多模块构建的多个模块，还有它们的依赖，以及可能还有除此之外的其它文件和构件。分发包的目的是将所有不同的代码，资源，文件包裹到一个单独的归档中，供用户方便的下载，拆解，运行。然而，我们也提到了一些使用套件描述符moduleSets元素的缺点——那就是POM之间的父子关系在某些情况会使子模块构件不可用。

具体的说，如果子模块POM引用的父项目包含了一段Assembly插件配置，在多模块构建运行的时候，这个父项目会在子模块之前被构建。但是父项目运行Assembly的时候会去寻找子模块的构件，而子模块项目还在等待父项目完整构建，这就造成了一个僵持的局面，父项目也就无法成功构建（因为它无法找到需要的子模块构件）。换句话说，子项目依赖于父项目，而父项目同时又依赖于子项目。

作为一个例子，考虑如下的套件描述符，在一个多模块结构的顶层项目中使用：

```
<assembly>
  <id>distribution</id>
  <formats>
    <format>zip</format>
    <format>tar.gz</format>
    <format>tar.bz2</format>
  </formats>

  <moduleSets>
```

```
    <moduleSet>
      <includes>
        <include>*-web</include>
      </includes>
      <binaries>
        <outputDirectory>/</outputDirectory>
        <unpack>true</unpack>
        <includeDependencies>true</includeDependencies>
        <dependencySets>
          <dependencySet>
            <outputDirectory>/WEB-INF/lib</outputDirectory>
          </dependencySet>
        </dependencySets>
      </binaries>
    </moduleSet>
    <moduleSet>
      <includes>
        <include>*-addons</include>
      </includes>
      <binaries>
        <outputDirectory>/WEB-INF/lib</outputDirectory>
        <includeDependencies>true</includeDependencies>
        <dependencySets>
          <dependencySet/>
        </dependencySets>
      </binaries>
    </moduleSet>
  </moduleSets>
</assembly>
```

基于这个父项目——叫做app-parent——它带有三个模块，分别为app-core，app-web，和app-addons，注意当我们运行这个多模块构建的时候发生了什么：

```
$ mvn package
[INFO] Reactor build order:
[INFO]   app-parent <----- PARENT BUILDS FIRST
[INFO]   app-core
[INFO]   app-web
[INFO]   app-addons
[INFO] ---------------------------------------------------------------
[INFO] Building app-parent
[INFO]    task-segment: [package]
[INFO] ---------------------------------------------------------------
[INFO] [site:attach-descriptor]
[INFO] [assembly:single {execution: distro}]
[INFO] Reading assembly descriptor: src/main/assembly/distro.xml
```

```
[INFO] ------------------------------------------------------------------------
[ERROR] BUILD ERROR
[INFO] ------------------------------------------------------------------------
[INFO] Failed to create assembly: Artifact:
org.sonatype.mavenbook.assemblies:app-web:jar:1.0-SNAPSHOT (included by module) doe
  an artifact with a file. Please ensure the package phase is run before the assemb
  generated.
...
```

父项目——app-parent——首先构建。这时因为所有其它项目都将其引用成父项目，从而使得它必须在所有其它项目之前构建。而app-web模块，套件描述符中第一个要处理的模块，还没有被构建。因此，它就没有相关的构件，因此整个装配就无法成功。

对于这种情形的一个临时解决办法是从Assembly插件声明中移除executions部分配置，这部分配置将插件绑定到父POM的package生命周期阶段，保持其它的配置不变。然后，分别运行两个Maven命令行任务：首先，package，构件整个多模块项目，然后，再运行assembly:assembly，直接调用Assembly插件，这时能够使用到之前运行输出的构件，并创建出分发套件。这样一个构建的命令行如下：

```
$ mvn package assembly:assembly
```

然而，这种方式有几个缺陷。首先，它使得套件分发过程变成了一个更手动的工作，对于整个构建过程，很大程度上这增加了复杂度，且会带来一些潜在的错误。此外，这也意味着如果不使用文件系统引用，那么附属构件——当项目构建运行的时候关联在内存中——也就无法再次被使用。

除了使用moduleSet来收集多模块构建中的所有构件，通常还有一种更好的的做法：使用一个专门的分发包项目模块，并利用模块间依赖。在这种方式中，你首先在你的构建中创建一个专门来出来装配分发包的模块。这个模块的POM包含了对所有其它模块的依赖，且配置了Assembly插件绑定到其package生命周期阶段。而套件描述符就使用dependencySets元素来取代moduleSets，以收集所有模块构件并决定它们在套件归档中的位置。这种方式避开了前面讲述的关于父子关系的陷阱，现在只需要在套件描述符中使用更简单的配置来完成我们的工作。

为此，我们可以创建一个新的项目结构，它与我们之间模块集合使用的方式类似，只是需要一个额外的分发项目，我们最终需要五个POM：app-parent，app-core，app-web，app-addons，以及app-distribution。新的app-deistribution的POM如下：

```
<project>
  <parent>
    <artifactId>app-parent</artifactId>
    <groupId>org.sonatype.mavenbook.assemblies</groupId>
    <version>1.0-SNAPSHOT</version>
  </parent>
```

```
  <modelVersion>4.0.0</modelVersion>
  <artifactId>app-distribution</artifactId>
  <name>app-distribution</name>

  <dependencies>
    <dependency>
      <artifactId>app-web</artifactId>
      <groupId>org.sonatype.mavenbook.assemblies</groupId>
      <version>1.0-SNAPSHOT</version>
      <type>war</type>
    </dependency>
    <dependency>
      <artifactId>app-addons</artifactId>
      <groupId>org.sonatype.mavenbook.assemblies</groupId>
      <version>1.0-SNAPSHOT</version>
    </dependency>
    <!-- Not necessary since it's brought in via app-web.
    <dependency> [2]
      <artifactId>app-core</artifactId>
      <groupId>org.sonatype.mavenbook.assemblies</groupId>
      <version>1.0-SNAPSHOT</version>
    </dependency>
    -->
  </dependencies>
</project>
```

注意由于现在的POM中不再能够使用模块集合，我们必须包含项目结构中对于其它模块的依赖。还要注意的是，我们这里没有显式的声明对于app-core的依赖，这时由于我们已经依赖的app-web，app-core已经通过传递性依赖的方式引入。

接着，当我们将distro.xml套件描述符移到app-distribution项目中时，我们需要使用dependencySets元素对其进行修改，如：

```
<assembly>
  ...
  <dependencySets>
    <dependencySet>
      <includes>
        <include>*-web</include>
      </includes>
      <useTransitiveDependencies>false</useTransitiveDependencies>
      <outputDirectory>/</outputDirectory>
      <unpack>true</unpack>
    </dependencySet>
    <dependencySet>
      <excludes>
```

```
        <exclude>*-web</exclude>
      </excludes>
      <useProjectArtifact>false</useProjectArtifact>
      <outputDirectory>/WEB-INF/lib</outputDirectory>
    </dependencySet>
  </dependencySets>
  ...
</assembly>
```

现在，如果我们从项目顶层运行构建，就能看到更好的结果：

```
$ mvn package
(...)
[INFO] ------------------------------------------------------------------
[INFO] Reactor Summary:
[INFO] ------------------------------------------------------------------
[INFO] module-set-distro-parent ...............SUCCESS [3.070s]
[INFO] app-core ............................. SUCCESS [2.970s]
[INFO] app-web .............................. SUCCESS [1.424s]
[INFO] app-addons ........................... SUCCESS [0.543s]
[INFO] app-distribution ..................... SUCCESS [2.603s]
[INFO] ------------------------------------------------------------------
[INFO] ------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------
[INFO] Total time: 10 seconds
[INFO] Finished at: Thu May 01 18:00:09 EDT 2008
[INFO] Final Memory: 16M/29M
[INFO] ------------------------------------------------------------------
```

正如你所看到的，使用依赖集合的方式更可靠，并且在运行构建的时候出错的几率更低——至少在Maven的内部项目构建排序逻辑还没有赶上Assembly插件的功能之前是这样。

12.7. 总结

如你在本章所看到的，Maven Assembly插件为创建自定义归档格式提供了很多可能性。虽然归档格式的细节可以十分复杂，但显然不是所有情况都这样，因此我们提供了一些内置的套件描述符。即使你的目标是以某种独特的目录结构来包含项目的依赖，选择项目文件，编写一个定制的套件描述符也不是太费力的事情。

套件对于很多应用都十分有用，但最有用的地方还是各种应用程序分发包。虽然使用Assembly插件有很多不同的方法，但在创建包含字节码的分发包时，使用标准化的套件描述符构件，同时避免使用moduleSets，是避免问题的两种很可靠的方法。

第 13 章 属性和资源过滤

13.1. 简介

本书的自始至终，你都会注意到我们可以在POM文件中使用属性引用。在多模块项目构建中我们可以使用org.sonatype.mavenbook和0.6-SNAPSHOT属性引用兄弟依赖，并且POM中的任意部分都可以由一个前缀为"project."的变量名引用。环境变量和Java系统变量也可以被引用，还包括在你~/.m2/settings.xml文件中的值。然而你还没有看到一个所有可能属性值的完整列表，以及如何使用它们帮助你创建可移植性的构建。本章提供了这样的一个列表。

如果你从来没有在你的POM中使用属性引用，你也应该知道Maven有一个特性叫做资源过滤，它能让你替换存储在src/main/resources目录下资源文件中的属性引用。这个特性可以用来为特定的平台定制构建，将重要的构建变量提取到属性文件中，POM中，或者profile中。本章介绍资源过滤特性并简单讨论如何用该特性创建可移植的企业级构建。

13.2. Maven属性

你可以在pom.xml文件或者资源文件中使用属性，资源文件被会Maven Resource插件的过滤特性处理。一个属性永远包含在${和}中。例如，要引用project.version属性，就需要这样写：

```
0.6-SNAPSHOT
```

在任何Maven项目中都有一些隐式的属性，这些隐式的属性是：

project.*
: Maven的项目对象模型（POM）。你可以使用该project.*前缀来引用任何在Maven POM中的值。

settings.*
: Maven Settings。你使用该settings.*前缀来引用~/.m2/settings.xml文件中Maven Settings的值。

env.*
: 环境变量如PATH和M2_HOME可以使用env.*前缀来引用。

系统属性
: 任何可以通过System.getProperty()方法获取的属性都可以作为Maven属性被引用。

除了上述这些隐式属性，Maven POM，Maven Settings，或者Maven Profile可以有一组任意的，用户自定义的属性。下面的小节详细介绍一个Maven项目中各种可用的属性。

13.2.1. Maven项目的属性

当一个Maven项目属性被引用的时候，属性名就直接引用Maven项目对象模型（POM）的一个属性。具体的说，你正在引用类`org.apache.maven.model.Model`的一个属性，这个类以隐式项目变量的方式向外暴露。当你使用该隐式变量引用属性的时候，你实际上正在使用简单的点标记来引用`Model`对象的一个bean属性。例如，当你引用`0.6-SNAPSHOT`的时候，你实际上是在调用暴露出来的`Model`对象实例的`getVersion()`方法。

POM同样也在所有Maven项目中以`pom.xml`文档的形式展现。任何在Maven POM 中的东西都可以用属性来引用。关于POM结构的完整参考可以访问http://maven.apache.org/ref/2.0.9/maven-model/maven.html。下面的列表展示了一些Maven项目中常见的属性引用。

`project.groupId` 和 `project.version`
: 大型的，多模块构建的项目通常共享同样的`groupId`和`version`定义符。当你为共享同样`groupId`和`version`的模块声明相互依赖的时候，为它们使用属性引用是一个很好的办法：

```
<dependencies>
  <dependency>
    <groupId>org.sonatype.mavenbook</groupId>
    <artifactId>sibling-project</artifactId>
    <version>0.6-SNAPSHOT</version>
  </dependency>
</dependencies>
```

`project.artifactId`
: 一个项目的artifactId通常被用作发布包的名称。例如，在一个打包类型为WAR的项目中，你会想生成一个不带版本定义符的WAR文件。为此，你就需要在你的POM中引用`project.artifactId`，如：

```
<build>
  <finalName>content-zh</finalName>
</build>
```

`project.name` 和 `project.description`
: 项目的name和description对于文档来说是比较常用的属性引用。你只要简单的引用这些属性，而不需要担心你所有的站点文档是否维护了一致的简短描述信息。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

project.build.*
: 如果你试图在Maven中引用输出目录，你绝不应该使用如target/classes的字面量，而是应该使用属性来引用这些目录。

- project.build.sourceDirectory
- project.build.scriptSourceDirectory
- project.build.testSourceDirectory
- project.build.outputDirectory
- project.build.testOutputDirectory
- project.build.directory

sourceDirectory，scriptSourceDirectory，和testSourceDirectory提供了项目源码目录的访问方式。outputDirectory和testOutputDirectory则能让你访问Maven放置字节码和其它构建输出的目录。directory引用的目录就是包含上述两个输出目录的父目录。

其它项目属性引用
: 在一个POM中有数百的属性可以引用。关于POM结构的完整参考见http://maven.apache.org/ref/2.0.9/maven-model/maven.html。

关于Maven Model对象可用属性的完整列表，可以看一下maven-model项目的JavaDoc：http://maven.apache.org/ref/2.0.9/maven-model/apidocs/index.html。当你载入这JavaDoc之后，看一下Model类。根据这个Model类的JavaDoc，你应该能够找到你想要引用的POM属性。如果你想要引用构建的输出目录，可以使用Maven Model的JavaDoc，明白该输出目录是通过model.getBuild().getOutputDirectory()引用的；该方法调用会被翻译成Maven的属性引用/usr/local/hudson/hudson-home/jobs/maven-guide-zh-to-production/workspace/content-zh/target/classes。

Maven Model定义了POM的结构，要了解关于Maven Model模块的更多信息，可以访问Maven Model的项目页面：http://maven.apache.org/ref/2.0.9/maven-model。

13.2.2. Maven的Settings属性

你也可以引用任何Maven本地Settings文件的属性，该文件通常位于~/.m2/settings.xml。这个文件包含了用户特定的配置，如本地仓库的位置，以及由某个特定用户配置的服务器，profile，和镜像。

关于本地Settings文件和对应属性的完整参考可以访问http://maven.apache.org/ref/2.0.9/maven-settings/settings.html。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

13.2.3. 环境变量属性

环境变量可以通过env.*前缀引用。以下列表是一些有趣的环境变量：

env.PATH
: 包含了Maven运行的当前PATH。该PATH包含了一个用来查找可运行脚本和程序的目录列表，

env.HOME
: （在*nix系统中）这个变量指向了用户的home目录。但你更应该使用/home/hudson，而非这个变量。

env.JAVA_HOME
: 指向了Java安装目录。它要么指向JDK安装目录，要么或者JRE目录。但你更应该考虑使用/usr/lib/jvm/java-1.6.0-openjdk-1.6.0.0/jre而非这个属性。

env.M2_HOME
: 指向了Maven2安装目录。

虽然他们都是可用的，但如果你有选择，你更应该使用Java系统属性。如果你需要用户的home目录，应该使用/home/hudson而非/usr/local/hudson。这么做，你最后会有可移植性更好的构建，也更符合Java平台一次编写到处运行的承诺。

13.2.4. Java系统属性

Maven暴露所有来自于java.lang.System的属性。任何你能从System.getProperty()获取的属性都能以Maven属性的形式引用。下面的表格列出了所有可用的系统属性：

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

表 13.1. Java系统属性

系统属性	描述
java.version	Java运行环境版本
java.vendor	Java运行环境供应商
java.vendor.url	Java供应商URL
java.home	Java安装目录
java.vm.specification.version	Java虚拟机规格说明版本
java.vm.specification.vendor	Java虚拟机规格说明供应商
java.vm.specification.name	Java虚拟机规格说明名称
java.vm.version	Java虚拟机实现版本
java.vm.vendor	Java虚拟机实现供应商
java.vm.name	Java虚拟机实现名称
java.specification.version	Java运行环境规格说明版本
java.specification.vendor	Java运行环境规格说明供应商
java.specification.name	Java运行环境规格说明名称
java.class.version	Java类格式版本号
java.class.path	Java类路径
java.ext.dirs	扩展目录的路径
os.name	操作系统名称
os.arch	操作系统架构
os.version	操作系统版本
file.separator	文件分隔符（UNIX上是"/"，Windows上是"\"）
path.separator	路径分隔符（UNIX上是":"，Windows上是";"）
line.separator	行分隔符（在UNIX和Windows上都是"\n"）
user.name	用户帐户名称
user.home	用户home目录
user.dir	用户当前工作目录

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

13.2.5. 用户定义的属性

除了由POM，Maven Settings，环境变量，和Java系统属性提供的隐式变量，你还可以定义自己的专有属性。这类属性可以定义在POM或者Profile中。而这些在POM或者Maven Profile中设置的属性可以像任何其它Maven属性一样被引用。用户定义的属性可以在POM中引用，也可以由Maven Resource插件用来过滤资源。这里是一个例子，在Maven POM中定义了一些专有的属性。

例 13.1. POM中的用户定义属性

```
<project>
  ...
  <properties>
    <arbitrary.property.a>This is some text</arbitrary.property.a>
    <hibernate.version>3.3.0.ga</hibernate.version>
  </properties>
  ...
  <dependencies>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate</artifactId>
      <version>${hibernate.version}</version>
    </dependency>
  </dependencies>
  ...
</project>
```

上述例子定义了两个属性：arbitrary.property.a和hibernate.version。hibernate.version在依赖声明中被引用。在属性名中使用句号作为分隔符是Maven POM和Profile中的一个标准实践。使用句号作为分隔符没有什么特殊的地方；对Maven来说，“hibernate.version”只是一个用来获取属性值“3.3.0.ga”的一个映射键。下一个例子展示了如何在Maven POM中的一个profile中定义一个属性。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

例 13.2. POM的Profile中的用户定义属性

```
<project>
  ...
  <profiles>
    <profile>
      <id>some-profile</id>
      <properties>
        <arbitrary.property>This is some text</arbitrary.property>
      </properties>
    </profile>
  </profiles>
  ...
</project>
```

前面的例子展示了在Maven POM的profile中定义一个用户自定义属性的过程。要了解更多的关于用户定义属性和profile的信息，请看第 11 章构建Profile。

13.3. 资源过滤

你可以使用Maven来对项目资源进行变量替换。在资源过滤被激活的时候，Maven会扫描资源，寻找由${ 和}包围的Maven属性的引用。一旦它找到这些引用，它就会使用合适的值去替换它们，就像前一节中定义的属性可以在POM中引用一样。当你需要根据目标部署平台使用不同的配置来参数化一个构建的时候，这就非常有用。

通常一个在src/main/resources目录下的.properties文件或者XML文档会包含对外部资源的引用，如需要根据目标部署平台进行不同配置的数据库或网络地址。例如，一个从数据库读取数据的系统有一个XML文档，其包含了数据库的JDBC URL以及安全凭证。如果你在开发和产品环境使用不同的数据库。你可以选择使用一种技术如JNDI将配置信息从应用程序提取出来至应用服务器，或者你可以创建一个知道如何根据目标平台用不同的值替换变量的构建。

使用Maven资源过滤，你可以引用Maven属性，并且使用Maven Profile来为不同的部署环境定义不同的配置。为了展示这个功能，假设你有一个项目，它使用了Spring Framework来配置来自于Commons DBCP[1]项目的BasicDataSource。你的项目可能在src/main/resources中包含一个文件applicationContact.xml，它含有如例 13.3 “在资源中引用Maven属性”所示的XML。

[1] http://commons.apache.org/dbcp

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

例 13.3. 在资源中引用Maven属性

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

    <bean id="someDao" class="com.example.SomeDao">
        <property name="dataSource" ref="dataSource"/>
    </bean>

    <bean id="dataSource" destroy-method="close" class="org.apache.commons.dbcp.Bas
        <property name="driverClassName" value="${jdbc.driverClassName}"/>
        <property name="url" value="${jdbc.url}"/>
        <property name="username" value="${jdbc.username}"/>
        <property name="password" value="${jdbc.password}"/>
    </bean>
</beans>
```

你的程序会在运行的时候读取这个文件，而你的构建将会使用定义在你pom.xml中的值替换如jdbc.url和jdbc.username这样的属性。默认情况资源过滤是关闭的，那是为了避免非有意的资源过滤。要开启资源过滤，你需要使用POM中build元素下的resources子元素。例 13.4 “定义变量和激活资源过滤”展示了这样一个POM，它定义了在例 13.3 “在资源中引用Maven属性”中引用的变量，并且为所有位于src/main/resources下的资源激活了资源过滤。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

例 13.4. 定义变量和激活资源过滤

```
<project>
  ...
  <properties>
    <jdbc.driverClassName>com.mysql.jdbc.Driver</jdbc.driverClassName>
    <jdbc.url>jdbc:mysql://localhost:3306/development_db</jdbc.url>
    <jdbc.username>dev_user</jdbc.username>
    <jdbc.password>s3cr3tw0rd</jdbc.password>
  </properties>
  ...
  <build>
    <resources>
      <resource>src/main/resources</resource>
      <filtering>true</filtering>
    </resources>
  </build>
  ...
  <profiles>
    <profile>
      <id>production</id>
      <properties>
        <jdbc.driverClassName>oracle.jdbc.driver.OracleDriver</jdbc.driverClassName
        <jdbc.url>jdbc:oracle:thin:@proddb01:1521:PROD</jdbc.url>
        <jdbc.username>prod_user</jdbc.username>
        <jdbc.password>s00p3rs3cr3t</jdbc.password>
      </properties>
    </profile>
  </profiles>
</project>
```

四个变量在properties元素中定义，并且我们为src/main/resources下的资源开启了资源过滤。资源过滤默认是关闭的，为了激活它你必须为项目中的资源显式的设置filtering为true。默认关闭资源过滤的目的是为了防止构建中偶然发生的非有意的过滤。如果你使用例 13.3 “在资源中引用Maven属性”中的资源和例 13.4 “定义变量和激活资源过滤”中的POM构建一个项目，然后列出target/classes下的资源内容，你会看到那里的资源文件已经被过滤。

```
$ mvn install
...
$ cat target/classes/applicationContext.xml
...
    <bean id="dataSource" destroy-method="close" class="org.apache.commons.dbcp.Bas
        <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
        <property name="url" value="jdbc:mysql://localhost:3306/development_db"/>
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛


```
        <property name="username" value="dev_user"/>
        <property name="password" value="s3cr3tw0rd"/>
    </bean>
...
```

例 13.4 “定义变量和激活资源过滤”中的POM还在`profiles/profile`元素下定义了一个production profile，它使用适合产品环境中的值覆盖了默认的属性。在这个特别的POM中，数据库连接默认的值是针对于安装在开发者机器上的本地MySQL数据库。而在production profile激活时构建这个项目，Maven会使用不同的驱动类，URL，用户名，和密码，配置系统连接到产品环境的Oracle数据库。如果你使用例 13.3 “在资源中引用Maven属性” 中的资源和例 13.4 “定义变量和激活资源过滤”中的POM，并激活了production file，然后构建一个项目，最后列出target/classes中的资源内容，你会看到其包含了使用产品环境值过滤的资源。

```
$ mvn -Pproduction install
...
$ cat target/classes/applicationContext.xml
...
    <bean id="dataSource" destroy-method="close" class="org.apache.commons.dbcp.Bas
        <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver"/>
        <property name="url" value="jdbc:oracle:thin:@proddb01:1521:PROD"/>
        <property name="username" value="prod_user"/>
        <property name="password" value="s00p3rs3cr3t"/>
    </bean>
...
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

第 14 章 Maven和Eclipse: m2eclipse

14.1. 简介

Eclipse IDE是目前Java开发人群中使用得最广泛的IDE。Eclipse有一大堆的插件（请看http://www.eclipseplugincentral.com/），无数的组织在它之上开发他们自己的软件。显然，Eclipse无处不在。m2Eclipse[1]项目在Eclipse IDE中提供了对Maven的支持，本章，我们将会研究它提供的特性，以帮助你在Eclipse IDE中使用Maven。

14.2. m2eclipse

m2eclipse插件（http://m2eclipse.codehaus.org/）为Eclipse提供了Maven的集成。m2Eclipse同时也以挂钩的方式连接了Subclipse插件（http://subclipse.tigris.org/）和Mylyn插件（http://www.eclipse.org/mylyn/）的特性。Subclipse插件为m2eclipse提供了与Subversion仓库交互的能力，Mylyn插件为m2eclipse提供了与任务集中接口交互的能力，该接口能跟踪开发过程的上下文。m2clipse提供的一些特性包括：

- 创建和引入Maven项目
- 依赖管理和与Eclipse classpath的集成
- 自动下载和更新依赖
- 构件的Javadoc及源码解析
- 使用Maven Archetypes创建项目
- 浏览，搜索远程Maven仓库
- 通过自动更新依赖列表管理POM
- 从Maven POM具体化一个项目
- 从多个SCM仓库签出一个Maven项目
- 适配嵌套的多模块Maven项目至Eclipse IDE
- 与 Web Tools Project （WTP）集成
- 与 AspectJ Development Tools（AJDT） 集成

[1] http://m2eclipse.codehaus.org/

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

- 与 Subclipse 集成
- 与 Mylyn 集成
- 基于表单的 POM 编辑器
- 依赖图的图形化显示
- 依赖树和已解析依赖的 GUI 展现

在上述列表以外m2eclipse还有很多其它的特性，本章介绍一些更令人印象深刻的特性。让我们从安装e2eclipse插件开始。

14.3. 安装 m2eclipse 插件

要安装m2Eclipse插件，你需要符合一些先决条件。你需要运行Eclipse 3.2或更高版本，JDK 1.4或更高版本，你需要确认Eclipse是在JDK上运行而不是JRE。在你有了Eclipse和兼容的JDK之后，你需要安装两个Eclipse插件：Subclipse和Mylyn。

14.3.1. 安装前提条件

你可以在安装m2eclipse的时候安装这些前提条件的软件，只要为每个前提条件软件添加一个远程更新站点至Eclipse。要安装这些先决条件软件，找到Help → Software Updates → Find and Install...。选择这个菜单项会载入Install/Update对话框。选择“Search for new features to install”选项然后点击Next。你将会看到一个“Update sites to visit”的列表。点击New Remote Site...，然后为每一个新的前提条件添加一个新的更新站点。为每个插件添加新的更新站点然后确认新站点被选择了。在你点击Finish之后，Eclipse会让你选择插件组件以安装。选择你想要安装的组件，Eclipse会下载，安装及配置你的插件。

需要注意的是如果你正在使用Eclipse最新的版本Eclipse 3.4(Ganymede)，安装插件的过程可能会有点不一样。在Ganymede中，你需要选择Help → Software Updates...，它会载入"Software Updates and Add-ons"对话框。在这个对话框中，选择“Available Software”面板然后点击Add Site...，它会载入"Add Site" 对话框。输入更新站点的URL然后点击OK。在"Software Updates and Add-ons"对话框中会出现更新站点上可用的插件。你可以选择你想要安装的模块然后点击Install...按钮。Eclilpse会解析所选插件的所有依赖，然后要求你同意插件的许可证。在Eclipse安装了新的插件之后，它会征求你的允许以重启。

14.3.1.1. 安装 Subclipse

要安装Subclipse，使用下面的Eclipse插件更新站点。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

- Subclipse 1.2: http://subclipse.tigris.org/update_1.2.x

想要了解其它版本的Subclipse，以及关于Subclipse插件更多的信息，请访问Subclipse项目的web站点：http://subclipse.tigris.org/。

14.3.1.2. 安装 Mylyn

要安装集成了JIRA支持的Mylyn，添加Mylyn Extras的Eclipse更新URL，如果你的组织使用Atlassian's JIRA[2]来跟踪问题，你会需要这么做。使用下面的更新站点来安装Mylyn：

- Mylyn (Eclipse 3.3): http://download.eclipse.org/tools/mylyn/update/e3.3 [3]
- Mylyn (Eclipse 3.4): http://download.eclipse.org/tools/mylyn/update/e3.4
- Mylyn Extras (JIRA 支持): http://download.eclipse.org/tools/mylyn/update/extras

想了解关于Mylyn项目的更多信息，访问Mylyn项目的web站点：http://www.eclipse.org/mylyn/。

14.3.1.3. 安装 AspectJ Tools Platform (AJDT)

如果你正在安装m2eclipse的0.9.4版本，你可能同时也想要安装Web Tools Platform (WTP) 和 AspectJ Development Tools (AJDT)。使用如下的eclipse更新URL以安装AJDT。

- AJDT (Eclipse 3.3): http://download.eclipse.org/tools/ajdt/33/update
- AJDT (Eclipse 3.4): http://download.eclipse.org/tools/ajdt/34/dev/update

想要了解更多的关于AJDT项目的信息，请访问AJDT项目的web站点http://www.eclipse.org/ajdt/。

14.3.1.4. 安装 Web Tools Platform (WTP)

要安装Web Tools Platform (WTP)。使用如下的eclipse更新URL，或者直接在Discovery站点中寻找Web Tool Project，该站点应该已经在你的Eclipse远程更新站点列表中了。

- WTP: http://download.eclipse.org/webtools/updates/

---

[2] http://www.atlassian.com/software/jira/

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

关于更多的Web Tools Platform的信息，请访问Web Tools Platform项目的web站点http://www.eclipse.org/webtools/。

14.3.2. 安装 m2eclipse

一旦你已经安装好这些先决条件，你从如下的Eclipse更新URL安装m2eclipse插件：

- m2eclipse 插件: http://m2eclipse.sonatype.org/update/

如果你想要安装最新的该插件的快照开发版本，你应该使用如下的开发更新URL而非之前的URL。

- m2eclipse 插件 （开发快照）: http://m2eclipse.sonatype.org/update-dev/

要安装m2eclipse，只需要添加一个正确的更新站点。至Help → Software Updates → Find and Install...，选择这个菜单项后会载入Install/Update对话框。选择"Search for new features to install"选项然后点击Next。你将会看到一个"Update sites to visit"列表。点击New Remote Site...，然后添加m2eclipse的更新站点。确认这个新添加的站点被选中了。在你点击Finish之后，Eclipse会要求你选择要安装的组件。你选好之后Eclipse会自动下载，安装，和配置m2eclipse。

如果你已经成功安装了这个插件，当你打开Window → Preferences...的时候，你应该能够在一个选项列表中看到一个Maven选项。

14.4. 开启 Maven 控制台

在我们开始查看m2eclipse的特征之前，首先让我们开启Maven的控制台。通过访问Window → Show View → Console来打开控制台视图。然后点击控制台视图右手边的一个小箭头，然后选择Maven控制台，如下显示：

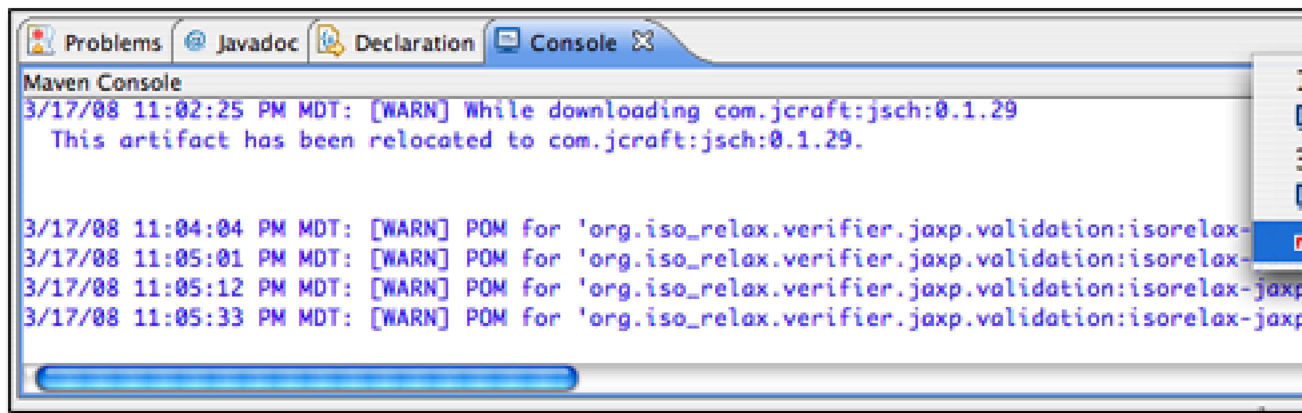


图 14.1. 在Eclipse中开启Maven控制台

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

Maven控制台显示那些当在命令行运行Maven时出现在控制台的Maven输出。能看到Maven正在干什么，以及根据调试输出来诊断问题，都是很实用的。

14.5. 创建一个 Maven 项目

在Maven中，我们使用archetype来创建项目。在Eclipse中，我们通过新建项目向导来创建项目。Eclipse中的新建项目向导为创建新项目提供了大量的模板。m2eclipse插件为这个向导增加如下的功能：

- 从SCM仓库签出一个Maven项目
- 使用Maven archetype创建一个Maven项目
- 创建一个Maven POM 文件

如 图 14.2 “使用m2eclipse向导来创建一个新项目”所示，这三个选项对使用Maven的开发人员来说都很重要。让我们逐个看一下。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

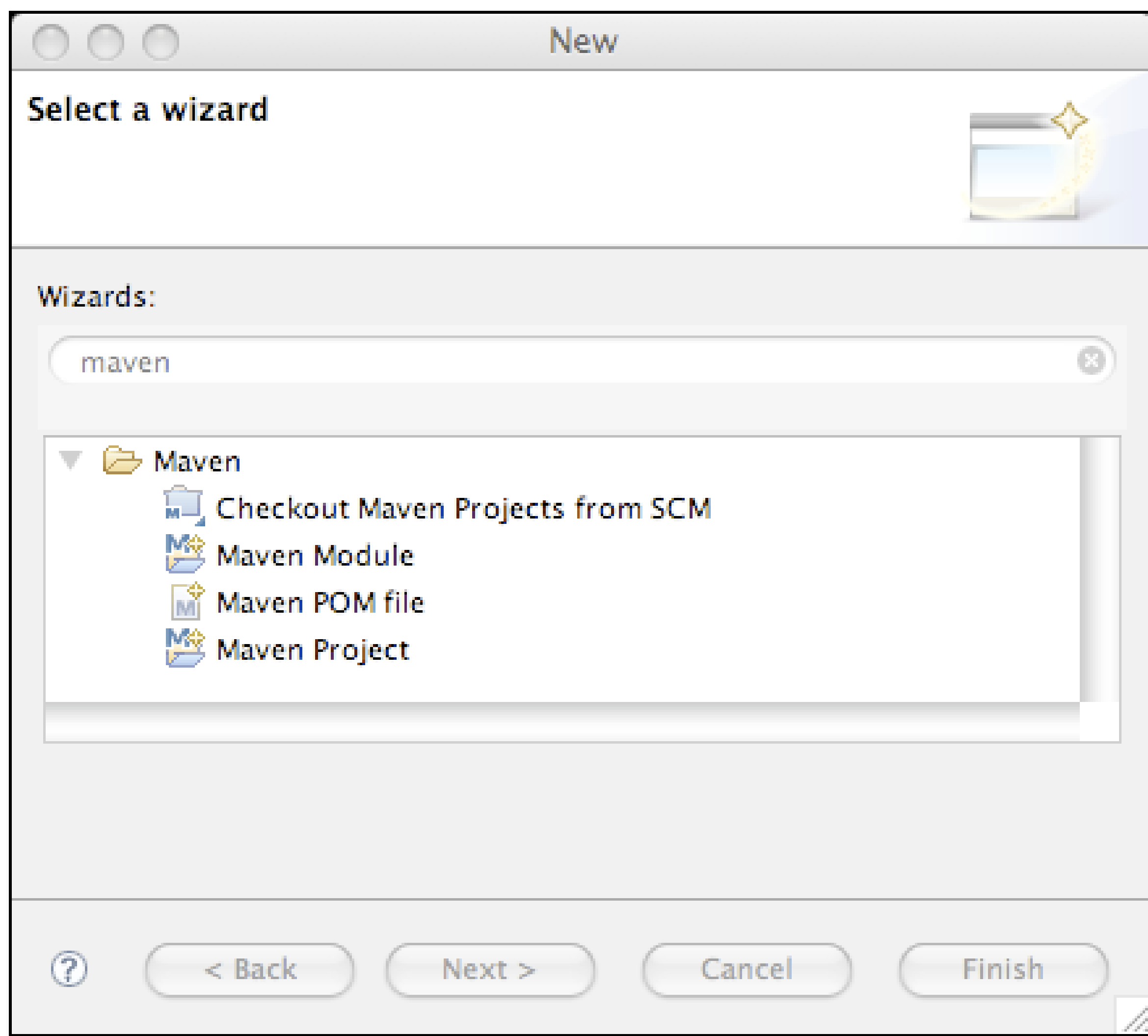


图 14.2. 使用m2eclipse向导来创建一个新项目

14.5.1. 从 SCM 签出一个 Maven 项目

m2eclipse提供了直接从SCM仓库签出项目的能力。简单的输入项目的SCM信息，它就会为你签出项目至你所选择的位置，如图 14.3 “从Subversion签出一个新的项目”：

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

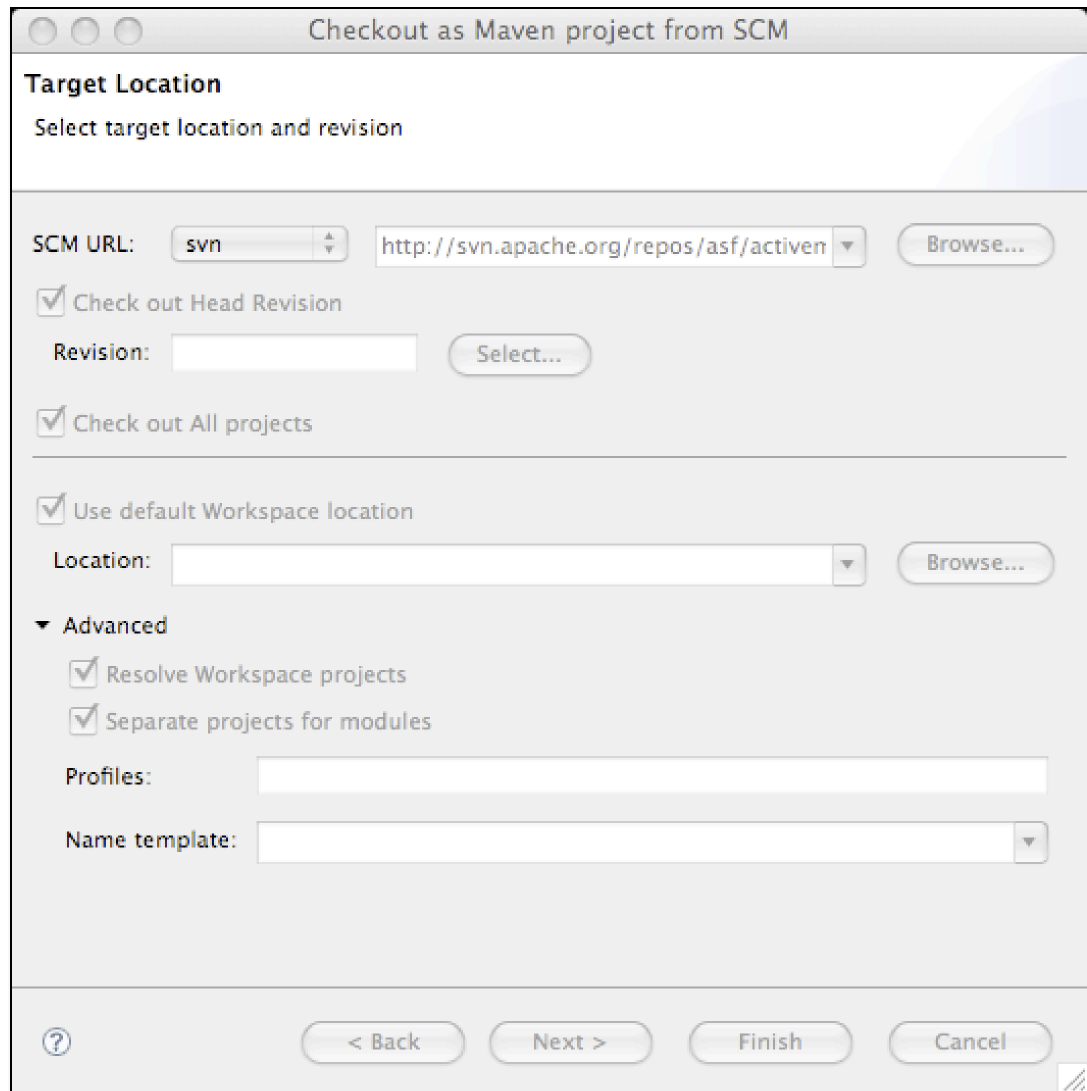


图 14.3. 从Subversion签出一个新的项目

该对话框中还有其它的选项用来浏览Subversion仓库的修订版以指定某个特定的修订版，或者直接手工输入修订版本号。这些特性重用了Subclipse插件的一些特性以和Subversion仓库相互。除了Subversion，m2eclipse插件也支持下面的SCM提供者：

- Bazaar
- Clearcase
- CVS
- git
- hg

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

- Perforce
- Starteam
- Subversion
- Synergy
- Visual SourceSafe

14.5.2. 用Maven Archetype创建一个Maven项目

m2eclipse提供了使用Maven Archetype创建一个Maven项目的能力。伴随着m2eclipse有许多可用的Maven Archetype，如图 14.4 “使用Maven Archetype创建一个Maven项目”：

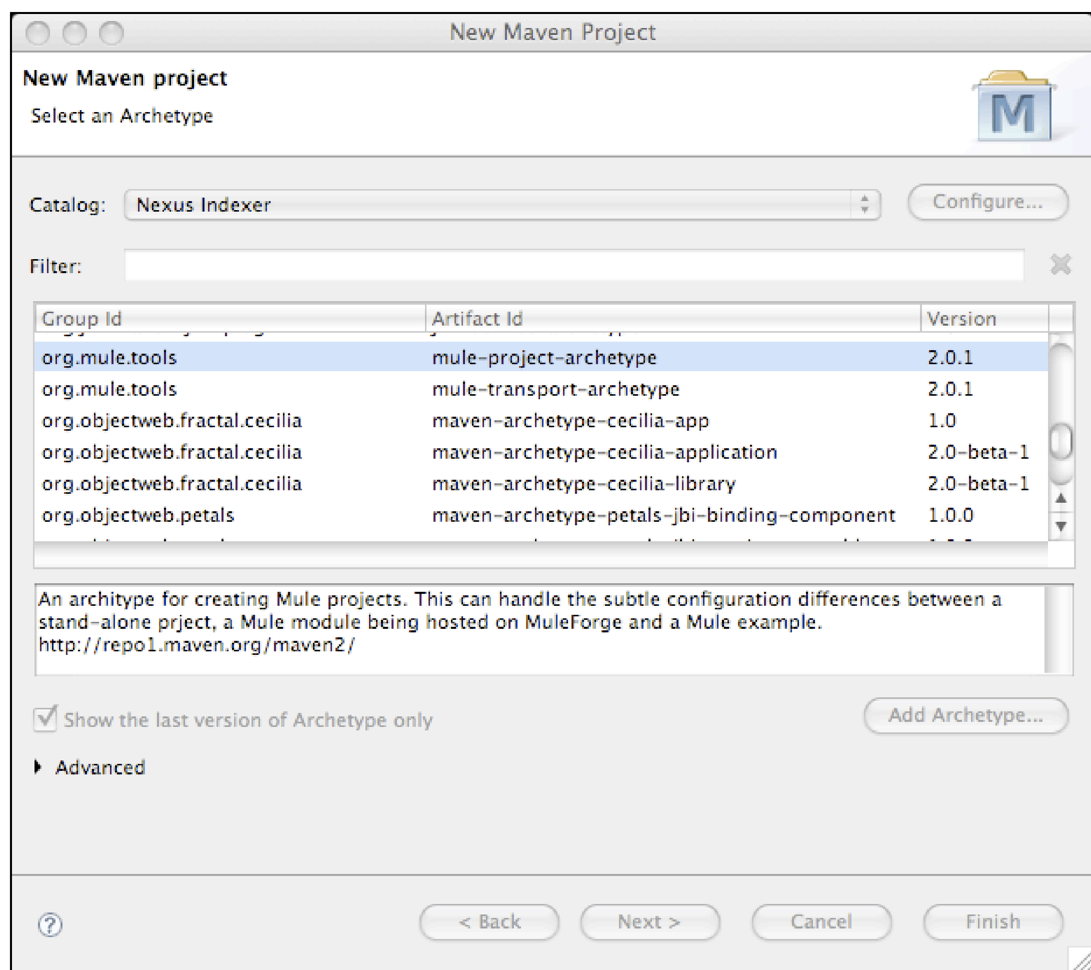


图 14.4. 使用Maven Archetype创建一个Maven项目

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

图 14.4 “使用Maven Archetype创建一个Maven项目”中的archetype列表是由一个叫Nexus索引器生成的。Nexus是一个仓库管理器，将会在第 16 章仓库管理器介绍。Nexus索引器是一个包含了整个Maven仓库索引的文件，m2eclipse使用它来罗列出所有Maven仓库中可用的archetype。到本章更新为止，m2eclipse大概在这个对话框中有90个archetype。其中比较突出的有：

- 标准的 Maven Archetypes 以创建
  - Maven插件
  - 简单Web应用
  - 简单项目
  - 新的Maven Archetypes
- Databinder[4] Archetype (数据驱动的 Wicket 应用程序) 位于 net.databinder
- Apache Cocoon[5] Archetype 位于 org.apache.cocoon
- Apache Directory Server[6] Archetype 位于 org.apache.directory.server
- Apache Geronimo[7] Archetype 位于 org.apache.geronimo.buildsupport
- Apache MyFaces[8] Archetype 位于 org.apache.myfaces.buildtools
- Apache Tapestry[9] Archetype 位于 org.apache.tapestry
- Apache Wicket[10] Archetype 位于 org.apache.wicket
- AppFuse[11] Archetype 位于 org.appfuse.archetypes
- Codehaus Cargo[12] Archetype 位于 org.codehaus.cargo
- Codehaus Castor[13] Archetype 位于 org.codehaus.castor
- Groovy-based Maven Plugin[14] Archetype (不推荐使用)[21] 位于 org.codehaus.mojo.groovy
- Jini Archetype
- Mule[15] Archetype 位于 org.mule.tools
- Objectweb Fractal[16] Archetype 位于 org.objectweb.fractal
- Objectweb Petals[17] Archetype 位于 org.objectweb.petals
- ops4j Archetype 位于 org.ops4j

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

- Parancoe[18] Archetype 位于 `org.parancoe`
- slf4j Archetype 位于 `org.slf4j`
- Springframework[19] OSGI 和 Web Services Archetype 位于 `org.springframework`
- Trails Framework[20] Archetype 位于 `org.trailsframework`

[21]这些只是由Nexus索引器目录罗列的archetype，如果你切换目录你会看到其它的archetype。虽然你看到的结果会有变化，但是以下额外的archetype能在Internal目录中得到：

- Atlassian Confluence[22] 插件 Archetype 位于 `com.atlassian.maven.archetypes`
- Apache Struts[23] Archetype 位于 `org.apache.struts`
- Apache Shale Archetype 位于 `org.apache.shale`

一个目录是对于仓库索引的简单引用。你看以通过点击在catalog下拉菜单旁边的Configure...按钮来管理一组m2eclipse已经了解的目录。如果你有你自己的archetype需要加入到这个列表中，可以点击Add Archetype...。

一旦你选择了一个archetype，Maven会从Maven仓库取得相应的artifact然后使用这个archetype创建一个新的Eclipse项目。

14.5.3. 创建一个 Maven 模块

m2eclipse提供了创建一个Maven模块的能力。创建一个Maven模块和创建一个Maven项目几乎一样，它也会用Maven archetype创建一个新的Maven项目。然而，一个Maven模块是另一个Maven项目的子项目，后者通常被认为是父项目。

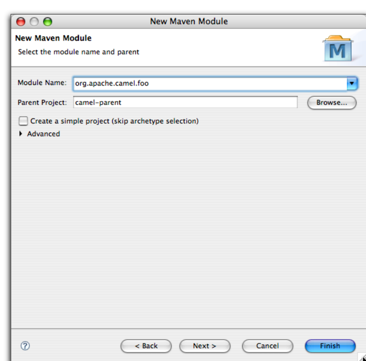


图 14.5. 创建一个Maven模块

[21]不要在Codehaus的Mojo项目中使用Groovy Maven插件。Jason Dillon已经将Groovy Maven集成移动到了codehaus的Groovy项目。更多的信息请访问http://groovy.codehaus.org/GMaven。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

当创建一个新的Maven模块的时候你必须选择一个在Eclipse中存在的父项目。点击浏览按钮，会看到一个已存在的项目的列表，如图 14.6 “为一个新的Maven模块选择一个父项目”：

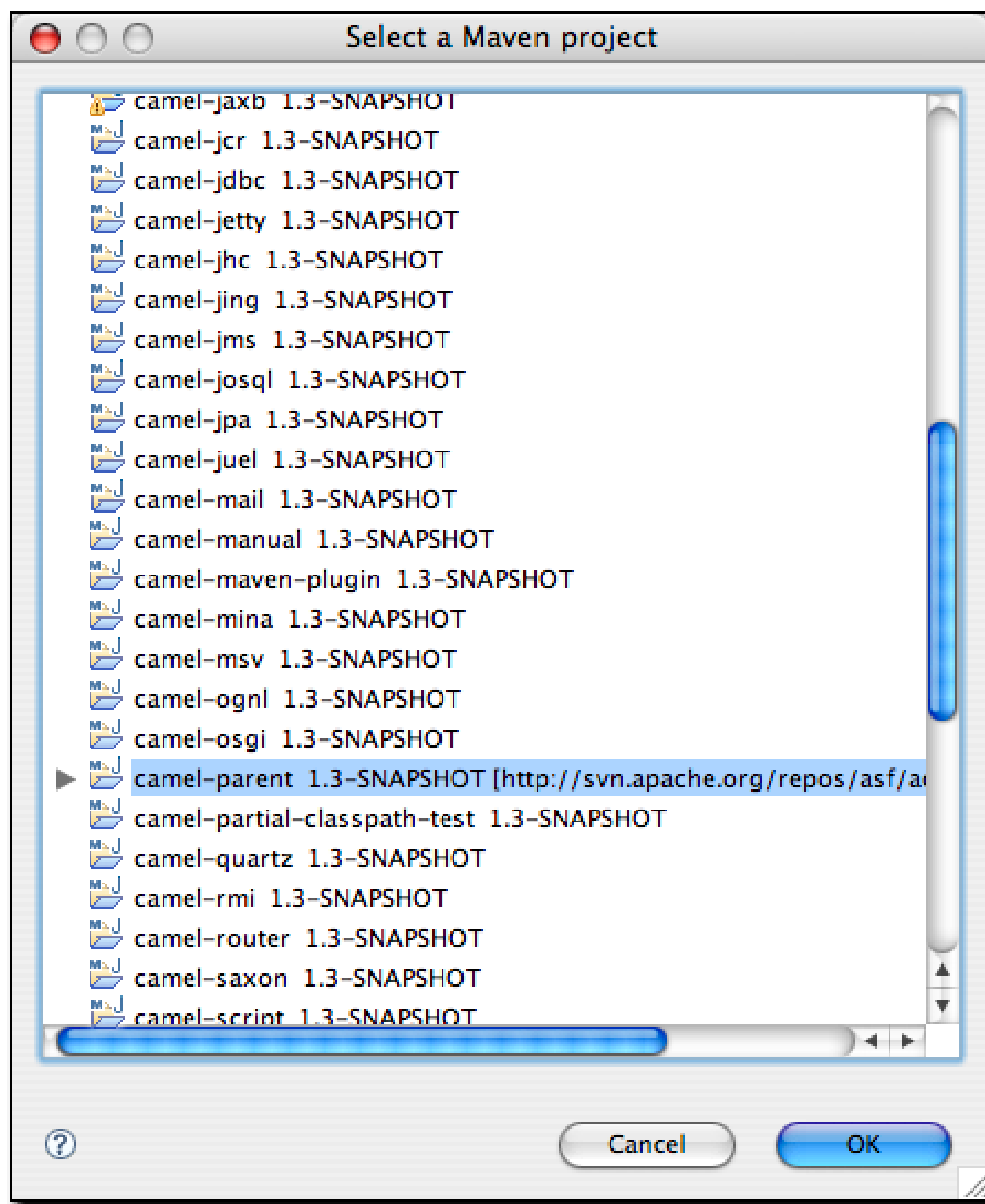


图 14.6. 为一个新的Maven模块选择一个父项目

在该列表中选择了一个父项目之后，你回到了创建新Maven模块的窗口，父项目字段已被填充，如图 14.5 “创建一个Maven模块”所示。点击Next你将会看到来

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

自第 14.5.2 节 “用Maven Archetype创建一个Maven项目的标准archetype列表，然后你可以选择用哪个archetype来创建Maven模块。

14.6. 创建一个Maven POM文件

另外一个m2eclipse提供的重要特性是它能创建一个新的Maven POM文件。m2eclipse提供了一个向导，可以用来很轻松的为一个已经在Eclipse中的项目创建一个新的POM文件。这个POM创建向导如图 14.7 “创建一个新的POM”所示：

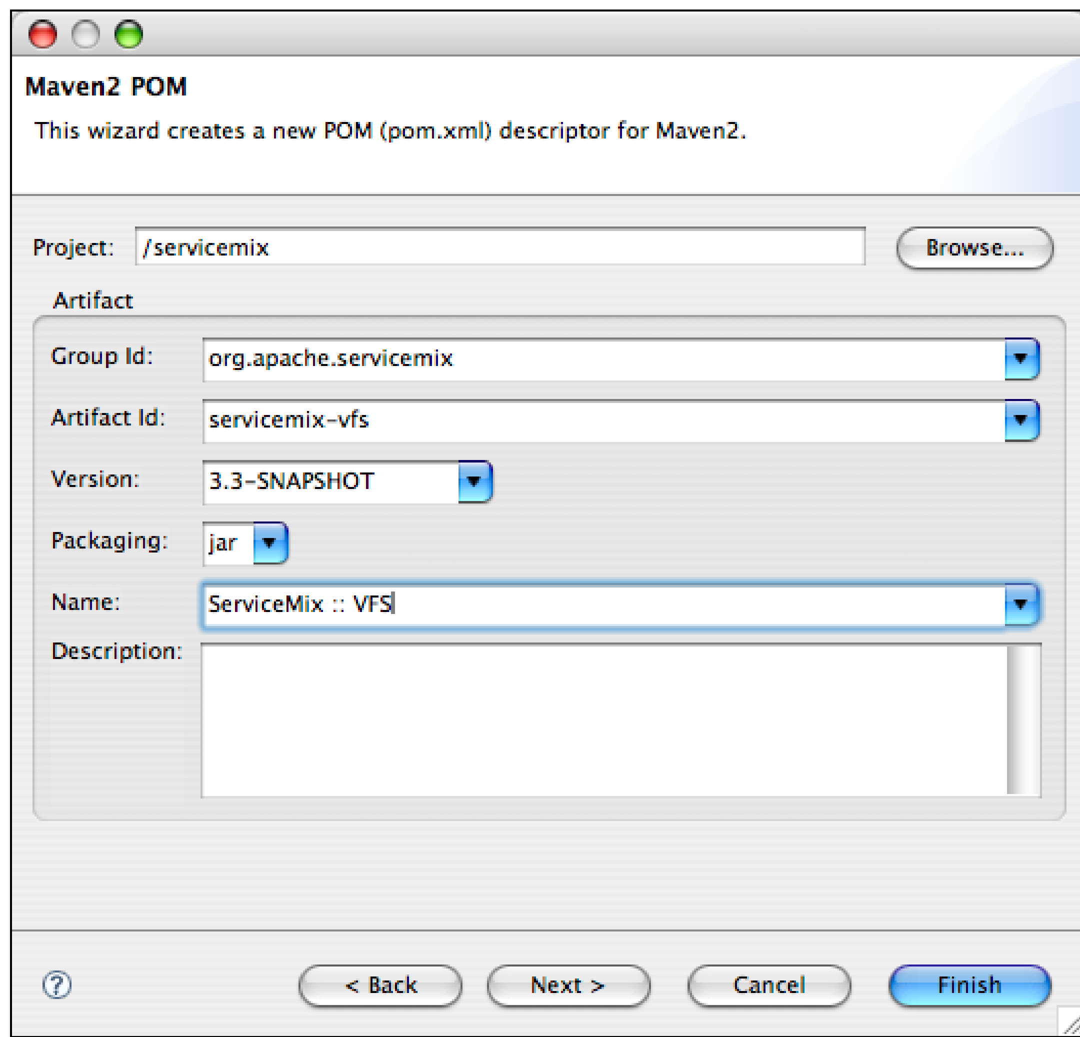


图 14.7. 创建一个新的POM

创建一个新的Maven POM大致就是选择一个项目，在m2eclipse提供的字段中输入Group Id,，Artifact Id,，Version，选择打包类型，以及提供一个名称。点击Next按钮开始添加依赖。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

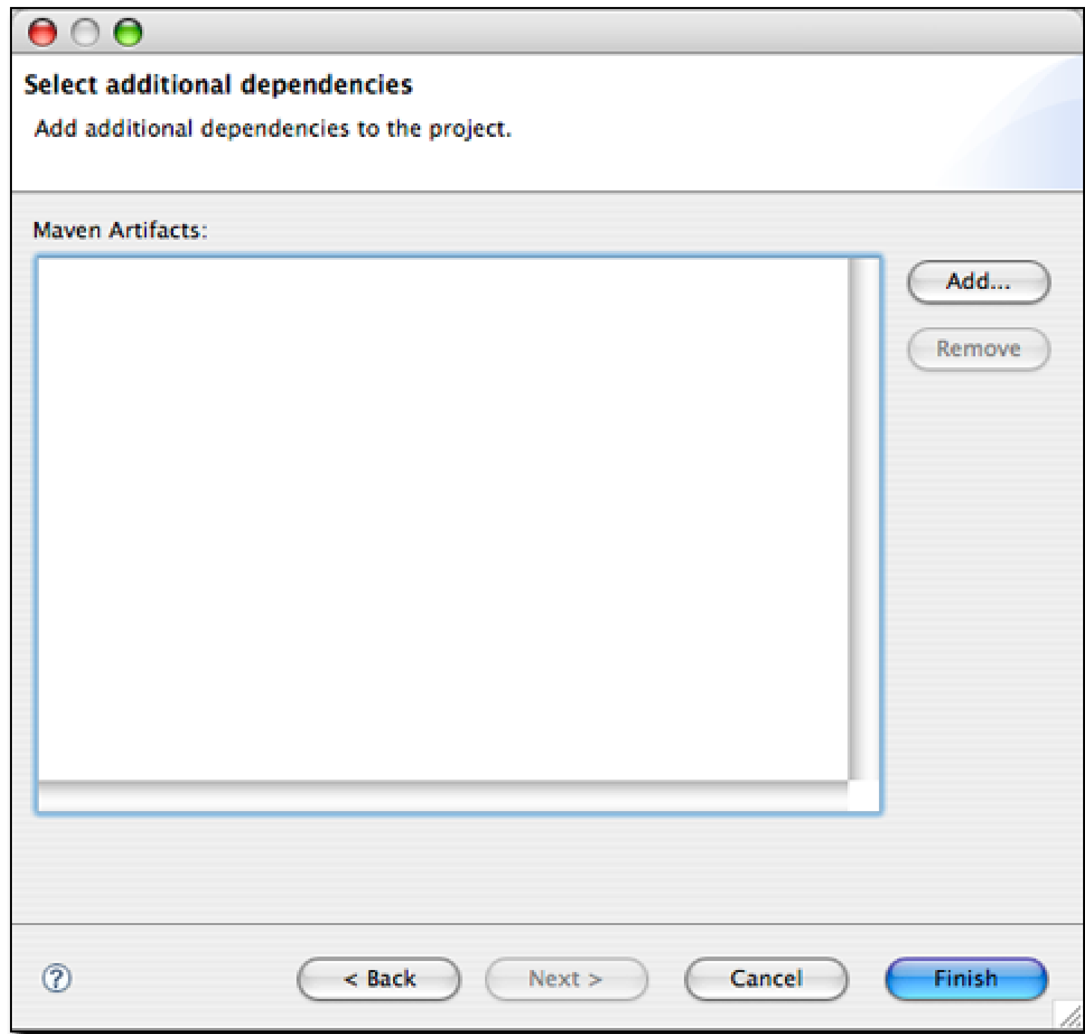


图 14.8. 为新的POM添加依赖

正如你能在图 14.8 “为新的POM添加依赖”看到的，POM中现在还没有依赖。点击Add按钮以向中央Maven仓库查询依赖，如图 14.9 “向中央仓库查询依赖”所示：

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

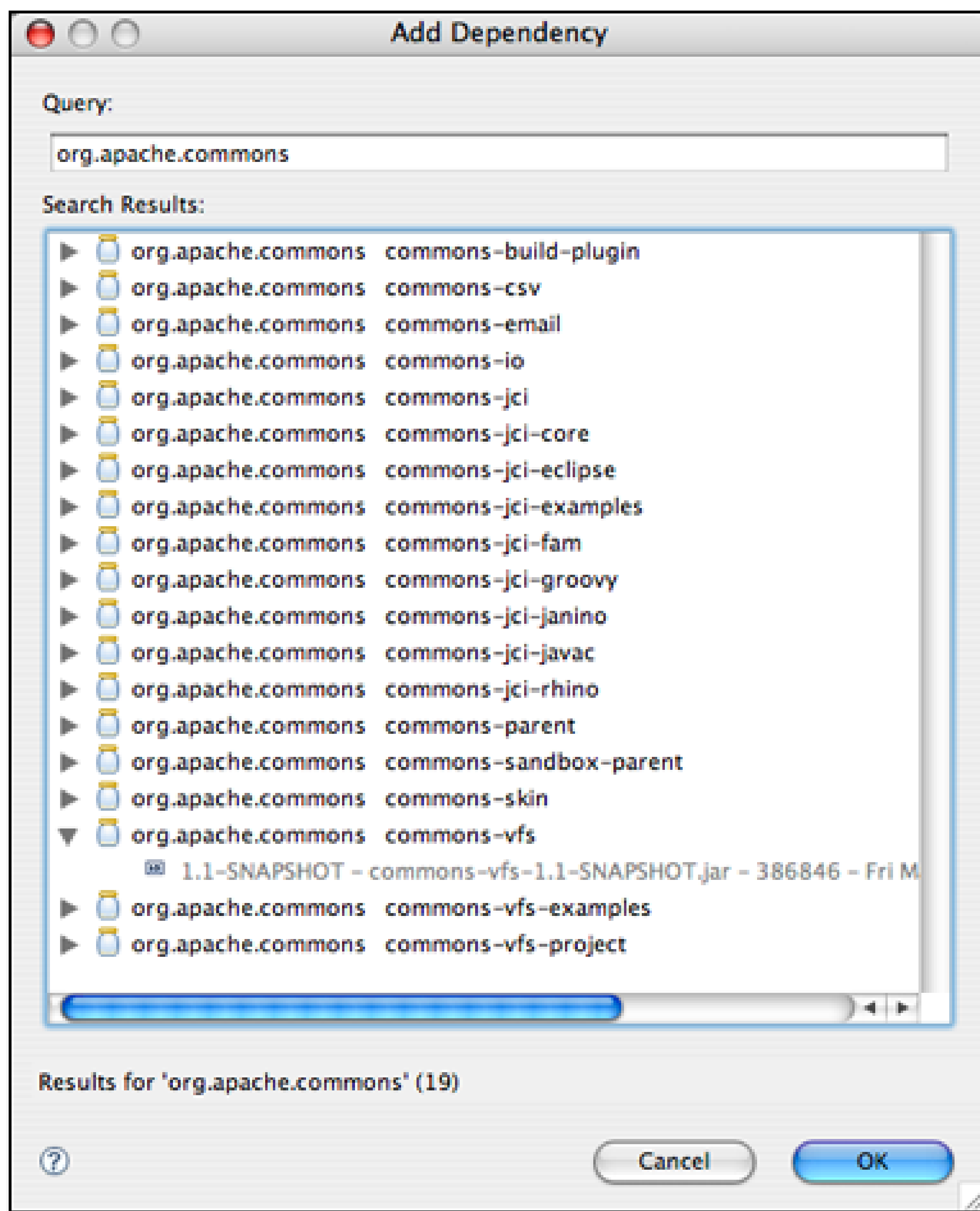


图 14.9. 向中央仓库查询依赖

查询依赖只是简单的输入你需要的构件的groupId。图 14.9 “向中央仓库查询依赖”展示了对org.apache.commons的一个查询，其中commons-vfs已被展开以查看可用的版本。选中commons-vfs的1.1-SNAPSHOT版本然后点击OK，你会回到依赖选择界面，你可以查询更多的构件或者直接点击finish按钮以创建POM。当你搜索依赖的时候，m2eclipse正使用在Nexus仓库管理器中使用的同样的Nexus仓库索引。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

现在你已经看到了m2eclipse创建新项目的特性，让我们看一下一组类似的将项目引入Eclipse的特性。

14.7. 导入Maven项目

m2eclipse为导入Maven项目至Eclipse提供了三种选择，分别是：

- 导入一个已存在的Maven项目
- 从SCM签出一个Maven项目
- 具体化一个Maven项目

图 14.10 “导入一个Maven项目”展示了m2eclipse提供的带有Maven选项的项目导入向导：

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

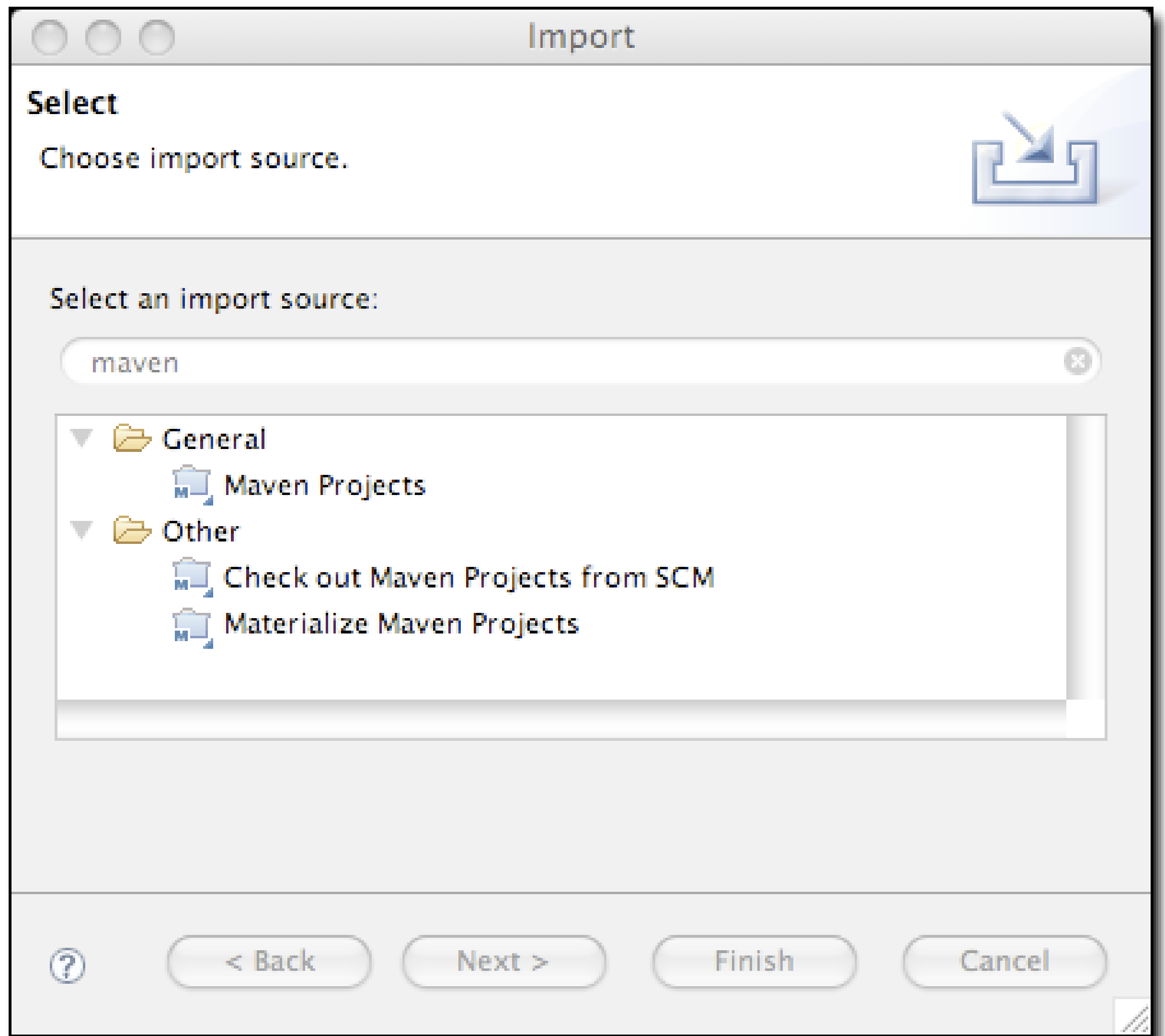


图 14.10. 导入一个Maven项目

使用Eclipse中的命令File → Import，然后在过滤字段中输入单词maven，就可以看到图 14.10 “导入一个Maven项目”的对话框。正如前面提到的，导入一个Maven项目至Eclipse有三种可用的方法：现存的Maven项目，从SCM签出一个项目，以及具体化Maven项目。

从Subversion导入一个Maven项目和前一节讨论的从Subversion创建一个Maven项目是等同的，因此再次讨论就显得冗余了。让我们往前走，看一下导入Maven项目至Eclipse的另外两个选项。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

14.7.1. 导入一个Maven项目

m2eclipse可以通过一个已存在的pom.xml导入一个Maven项目。通过指向Maven项目所在的目录，m2eclipse能探测到该项目中的所有POM，然后提供一个这些POM的层次列表，如图 14.11 “导入一个多模块的Maven项目”。

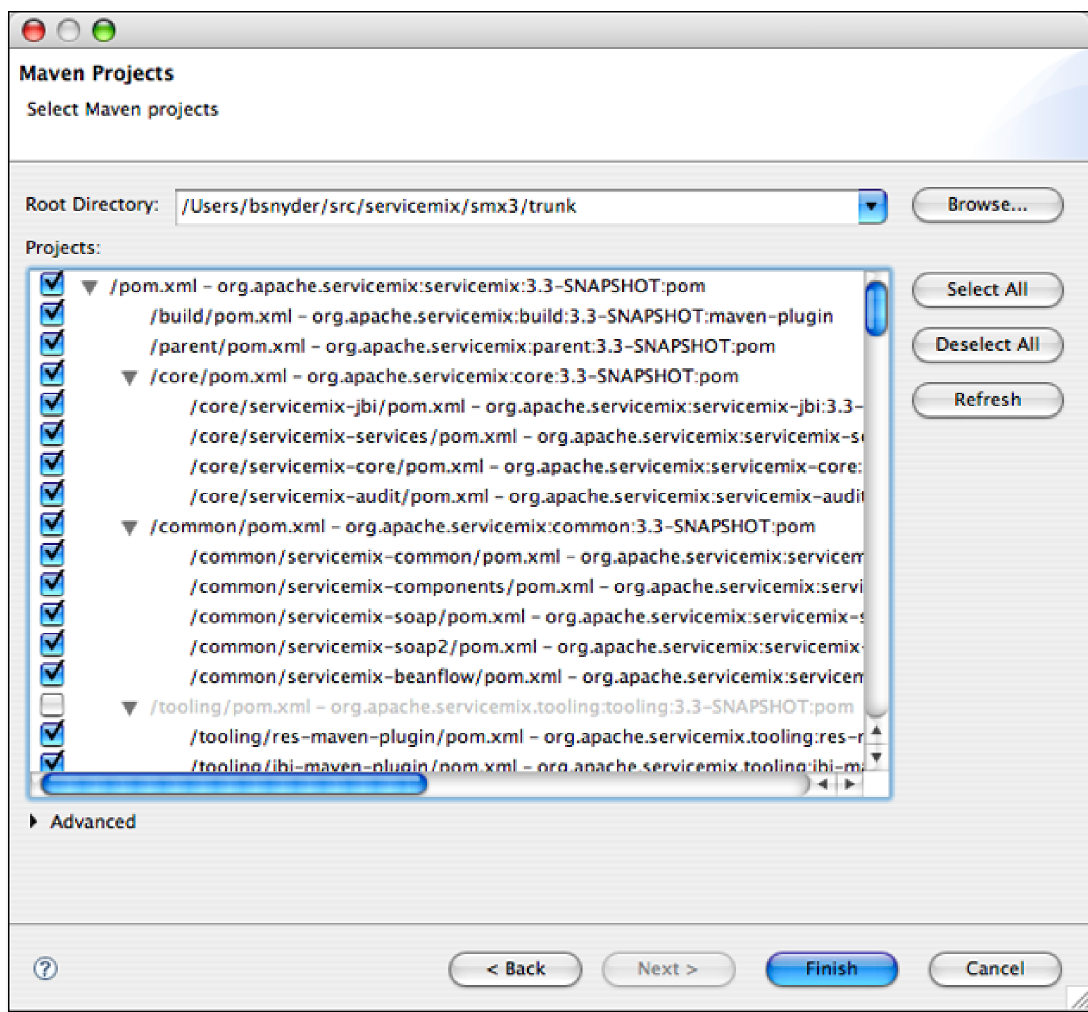


图 14.11. 导入一个多模块的Maven项目

图 14.11 “导入一个多模块的Maven项目”显示了被导入的项目的视图。注意该项目中所有的POM是分层的。这让你能够很简单的选择到你想要导入至Eclipse的POM（也就是你想要导入的项目）。当你选择了你想要导入的项目之后，m2eclipse会使用Maven导入并构建这个项目。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

14.7.2. 具体化一个Maven项目

Maven还提供了“具体化”一个Maven项目的能力。具体化类似于从Subversion签出一个Maven项目的过程，但此时Subversion URL是从项目的根POM文件找到的，而不是手工的输入。如果一个POM文件有正确的元素来指定源代码仓库的位置，你就能仅仅通过这个POM文件来“具体化”Maven项目。使用这个特性，你可以浏览中央Maven仓库中的项目，然后将其具体化成Eclipse项目。如果你的项目依赖于一个第三方的开源库，而且你需要查看这个库的源码，具体化的特性就变得十分方便和实用。现在只需要实用m2eclipse魔术般的“具体化”特性将项目导入到Eclipse中，而不是去追查项目的web站点然后寻找如何将其从Subversion签出。

图 14.12 “Materializing a Maven Project”展示了选择具体化Maven项目后的向导：

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

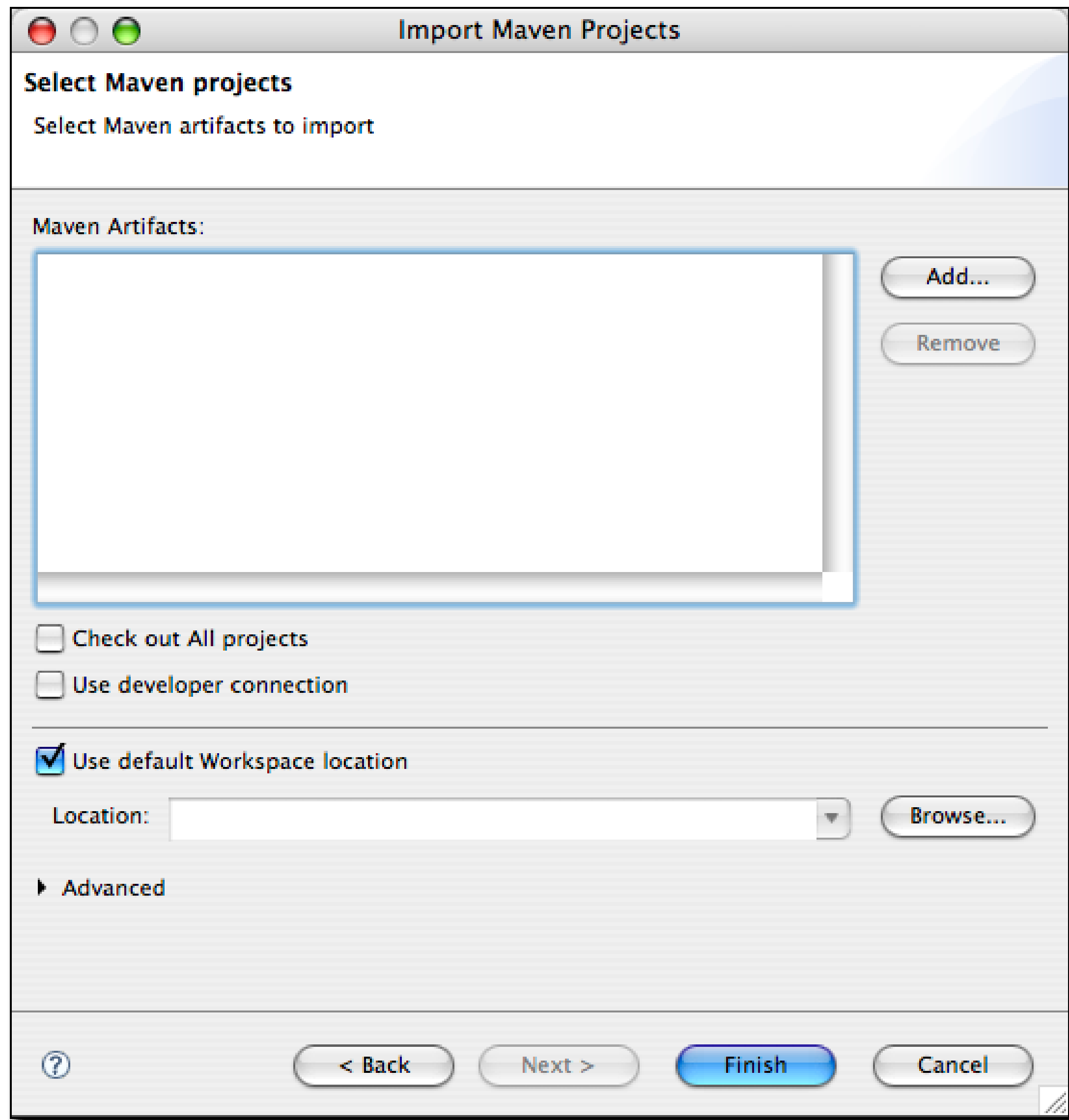


图 14.12. Materializing a Maven Project

注意在这个对话框中Maven artifacts是空的。这是因为还没有添加项目。为了添加一个项目，你需要点击右边的Add按钮然后选择一个来自中央Maven仓库的依赖以添加。图 14.13 “选择一个构件以具体化”展示了如何添加一个项目：

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

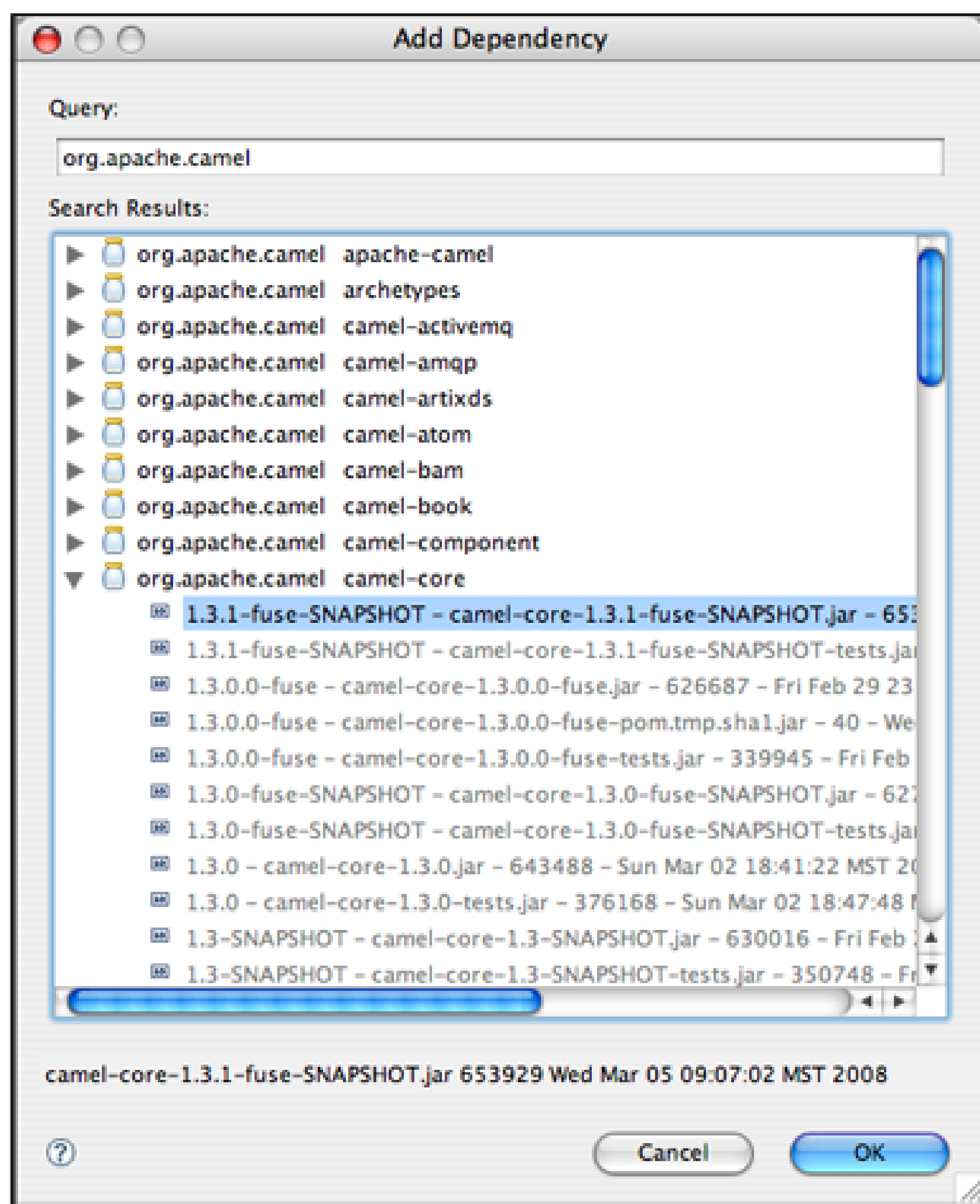


图 14.13. 选择一个构件以具体化

当输入查询的时候，候选的依赖将会被在本地Maven仓库找到。花几秒钟对本地Maven仓库索引之后，候选依赖列表就会显示。选择一个要添加的依赖然后点击OK，这样它们就会被添加到列表中如图 14.14 “具体化Apache Camel”。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

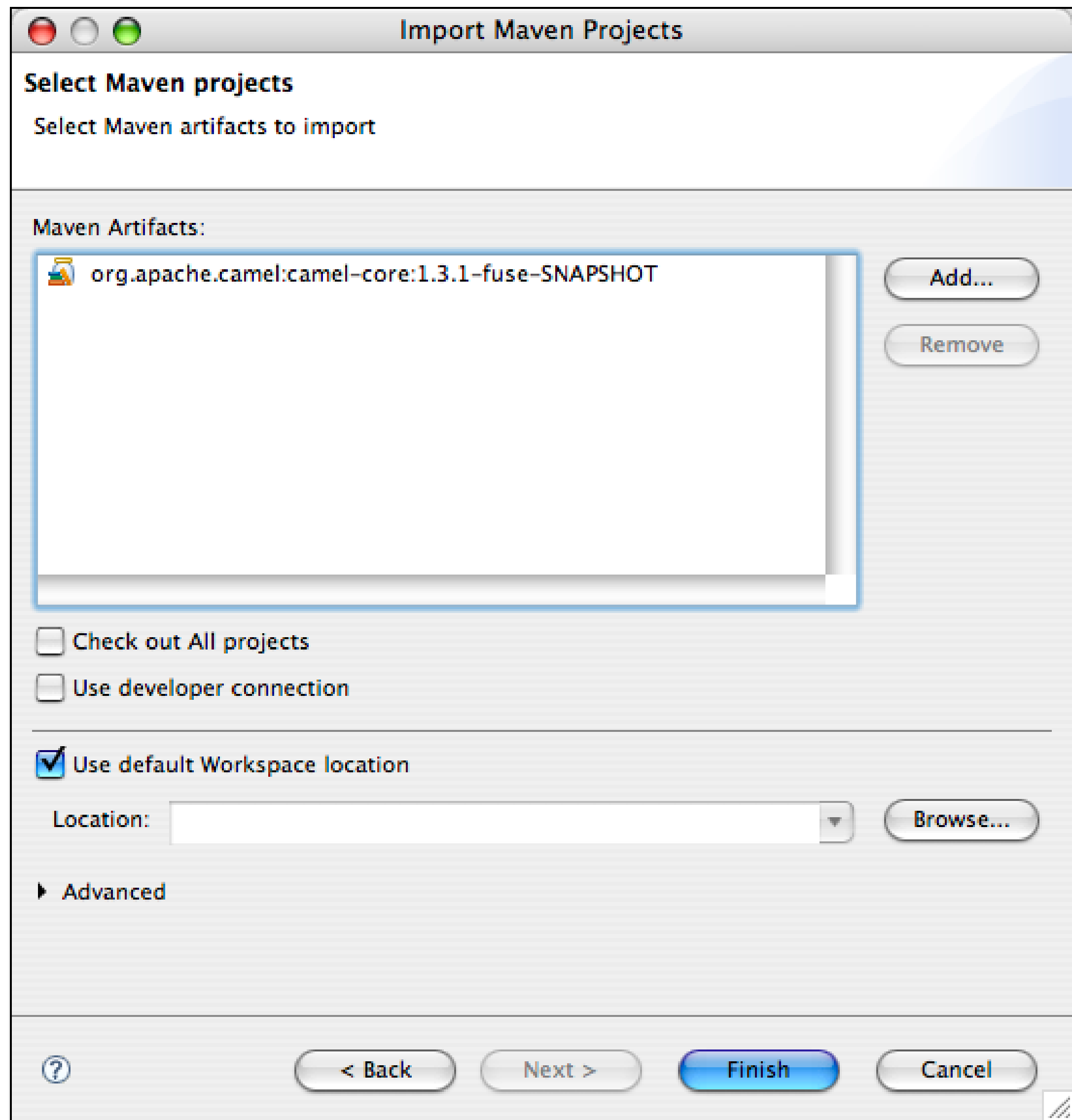


图 14.14. 具体化Apache Camel

在添加一个依赖的时候，你有一个选项，让m2eclipse签出这个构件的所有项目。

14.8. 运行Maven构建

m2eclipse修改了Run As...和Debug As...菜单，以让你能够在Eclipse中运行Maven。图 14.15 “通过Run As..运行一个Eclipse构建”展示了一个m2eclipse项目的Run As...菜单。从这个菜单你可以运行一些常用的生命周期过程如clean，install，或者package。你也可以载入运行配置对话框窗口，然后使用参数及更多的选项来配置一个Maven构建。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

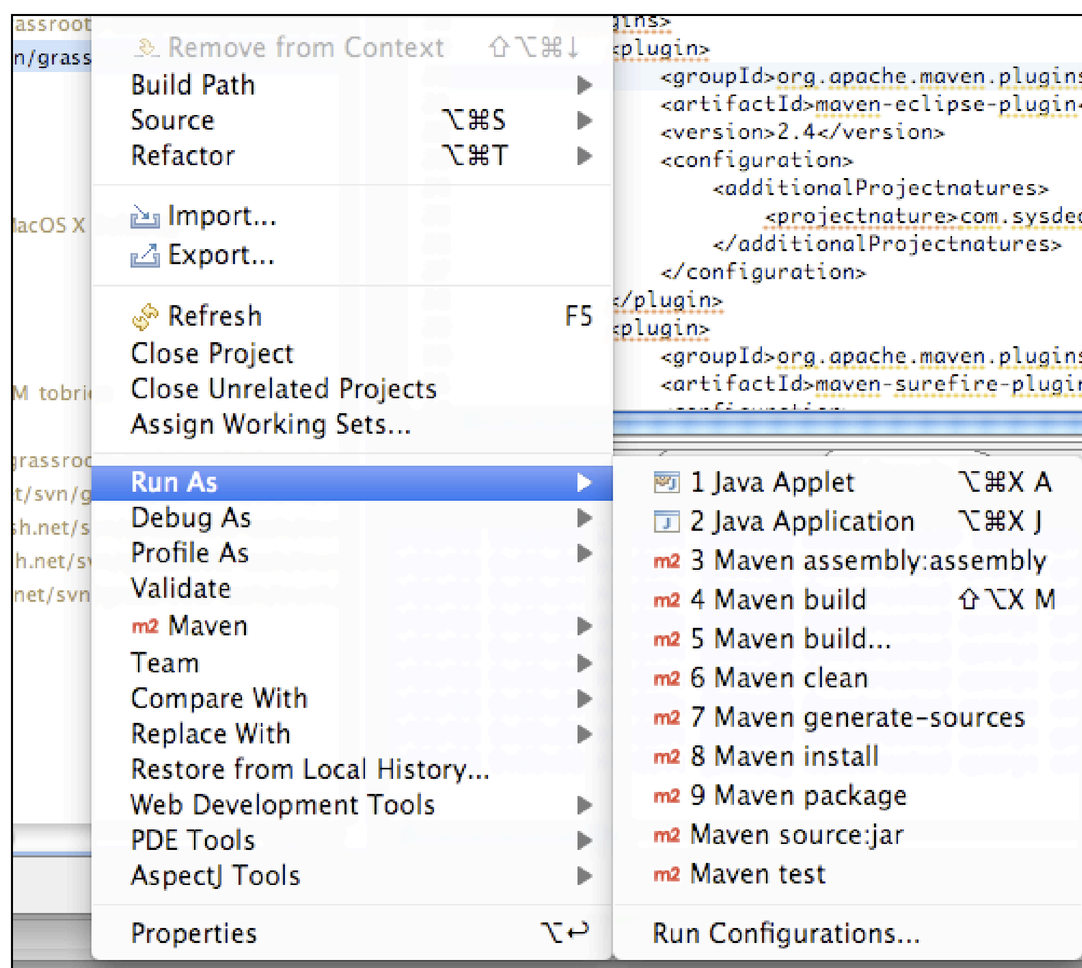


图 14.15. 通过Run As..运行一个Eclipse构建

如果你需要用更多的选项来配置一个Maven构建，你可以选择Run Configurations...然后创建一个Maven构建。图 14.16 “配置一个Maven构建作为一个运行配置”展示了配置一个Maven构建的运行配置对话框。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

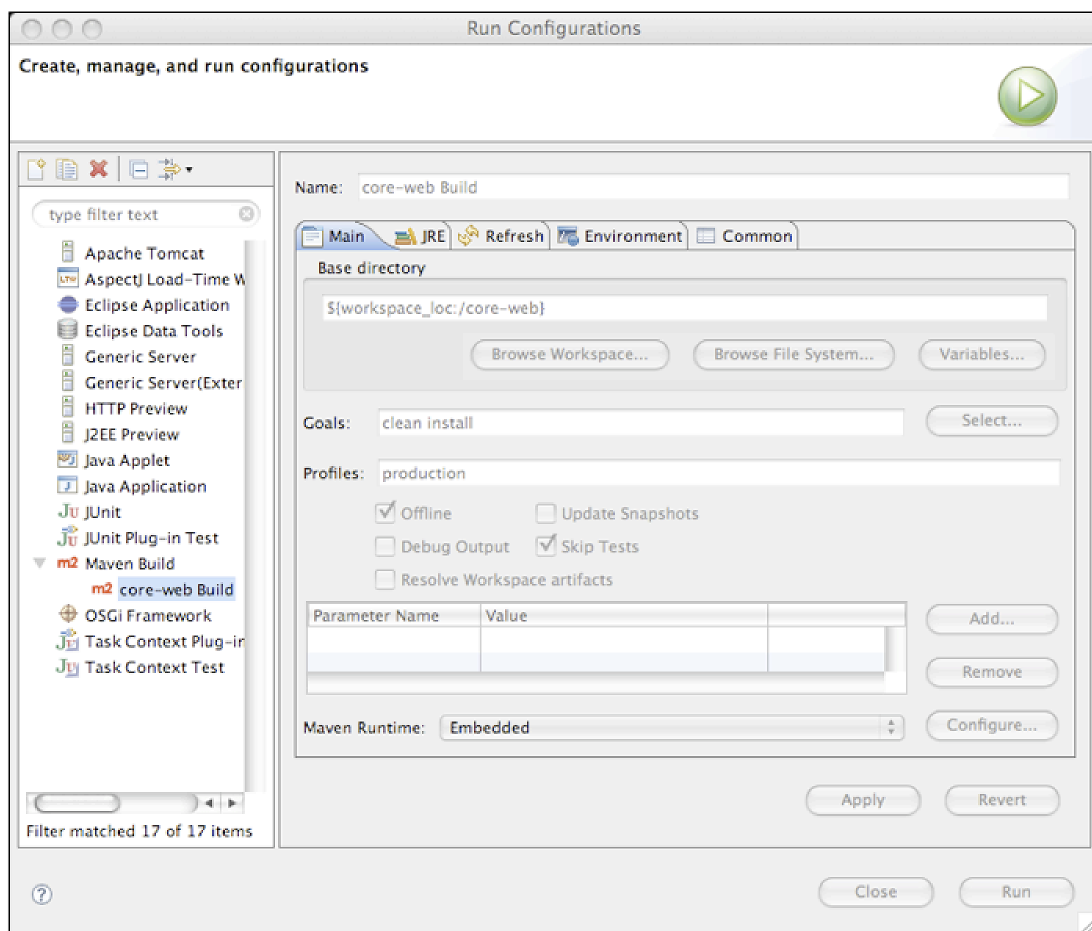


图 14.16. 配置一个Maven构建作为一个运行配置

运行配置对话框允许你指定多个目标和profile，它暴露了类似于“skip tests”和“update snapshots”的选项，并且允许你自定义从项目到JRE到环境变量的一切。你可以使用这个对话框来支持任何你希望在m2eclipse中启动的自定义Maven构建。

14.9. 使用Maven进行工作

当项目在Eclipse中的时候，m2eclipse插件为使用Maven提供了一组特性。有很多特性使得在Eclipse中使用Maven变得十分容易，让我们仔细看一下。在前一节，我们具体了一个Maven项目并且选择了一个来自于Apache Camel的名为camel-core的子项目。我们将使用这个项目来演示这些特性。

通过在camel-core项目上右击，然后选择Maven菜单项，你能看到可用的Maven特性。图 14.17 “可用的Maven特性”展示了这些特性的一个快照。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

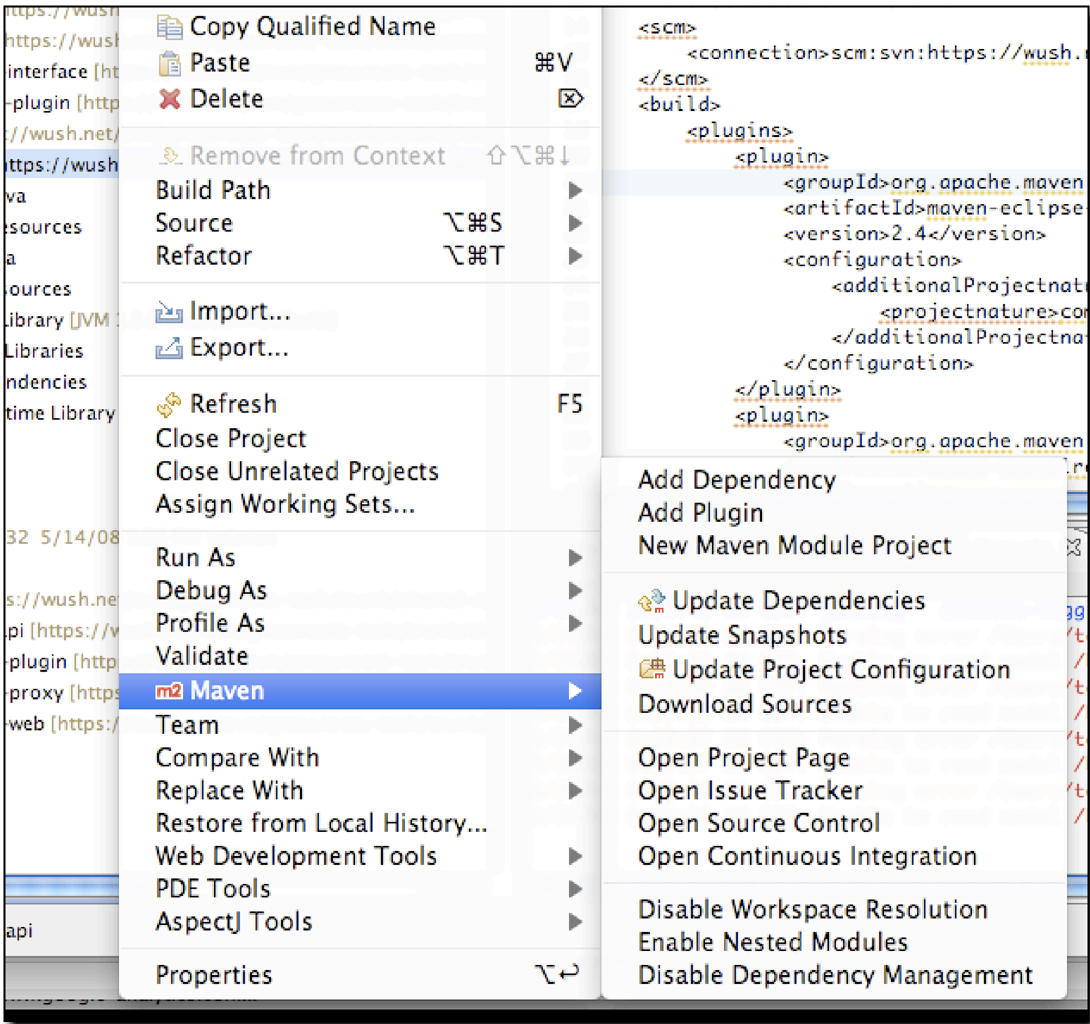


图 14.17. 可用的Maven特性

注意在图 14.17 “可用的Maven特性”中camel-core项目可用的特性包括：

- 添加依赖和插件
- 更新依赖，快照和源代码文件夹
- 创建一个Maven模块
- 下载源代码
- 打开项目的URL如项目Web页面，问题追踪系统，源码控制，和持续集成工具
- 开启/关闭工作台解析器，嵌套Maven模块和依赖管理

这些特性都能帮你节省很多时间，让我们先简单的看一下。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

14.9.1. 添加及更新依赖或插件

让我们假设我们想要给camel-core POM添加一个依赖或者一个插件。为了示范，我们会添加commons-lang作为一个依赖。（请注意添加依赖或者插件的功能完全一样，因此我们就用添加一个依赖作为示范。）

m2eclipse为给一个项目添加依赖提供了两种选项。第一种选项是通过手动的编辑POM文件的XML内容来添加一个依赖。这种手动编辑POM文件方式的缺点是你必须知道构件的信息，或者，你可以使用下一节讨论的特性来手工的定位仓库索引中的构件信息。好处是在你手工添加依赖并保存POM文件之后，项目的Maven依赖容器会自动更新以包含这个新的依赖。图 14.18 “手动给项目的POM添加一个依赖”展示了如何给camel-console POM 添加对commons-lang的依赖，然后Maven依赖容器自动更新并包含了这个依赖。

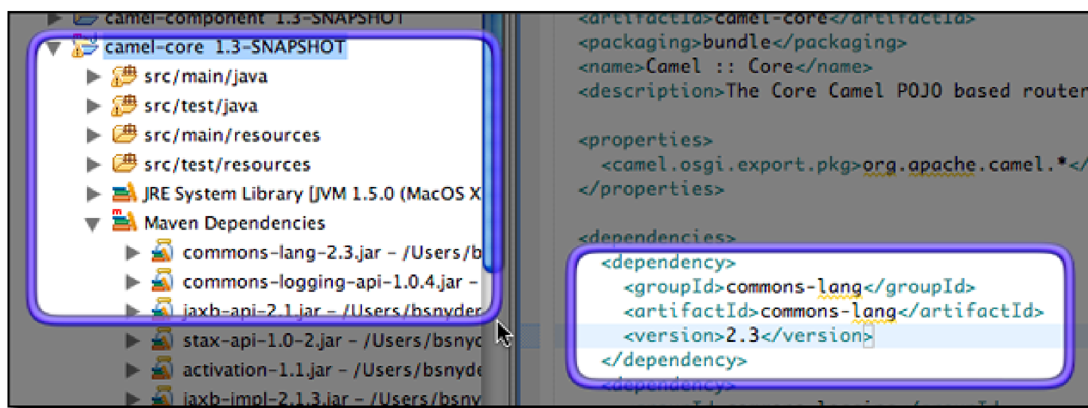


图 14.18. 手动给项目的POM添加一个依赖

手动添加依赖效果不错但是它比第二种方式需要更多的工作。在手动给POM添加依赖元素的时候，Eclipse工作台右下角的进程反映了这一动作，如图 14.19 “更新Maven依赖”：

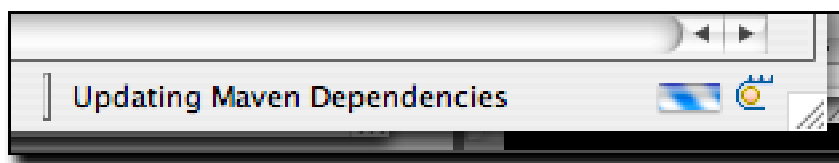


图 14.19. 更新Maven依赖

第二种添加依赖的方式容易得多，因为你不需要知道构件的除groupId以外的信息。图 14.20 “搜索依赖”展示了这种功能：

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

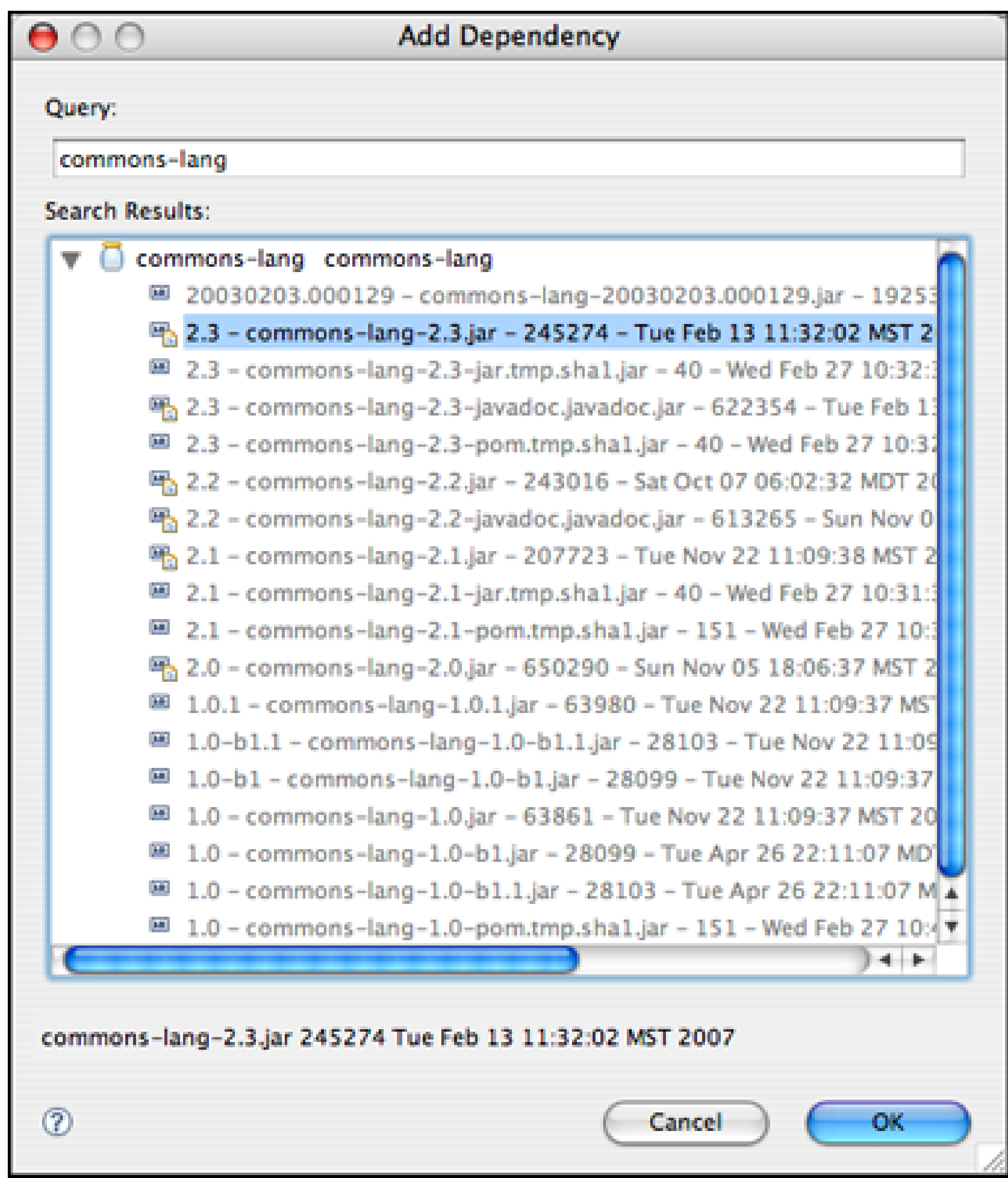


图 14.20. 搜索依赖

通过简单的在搜索框中输入信息，m2eclipse会查询仓库索引，显式在本地Maven仓库中构件的版本。这种方式更好因为它能节省大量的时间。有了m2eclipse，你不再需要中央Maven仓库中搜寻一个构件版本。

14.9.2. 创建一个Maven模块

m2eclipse使得在一个多模块的Maven项目中创建一系列的嵌套项目变得十分容易。如果你有一个父项目，而且你想给这个项目添加一个模块，只需要在项目上右击，打

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

开Maven菜单，选择“New Maven Module Project”。m2eclipse会带你创建一个新项目，之后他会更新父项目的POM以包含子模块的引用。在m2eclipse出现之前，很难在Eclipse中使用Maven项目的层次特性。有了m2eclipse，父子项目关系的底层细节被集成到了开发环境中。

14.9.3. 下载源码

如果中央Maven仓库包含了某个特定项目的源码构件，你可以从仓库下载这份源码然后在Eclipse环境中使用它。当你正在Eclipse中调试一个复杂的问题的时候，没有什么能比在Eclipse调试器中的第三方依赖上右击然后研究源码来的更方便的了。选择该选项之后，m2eclipse会尝试着从Maven仓库下载源码构件。如果不能取得源码构件，你应该去问项目的维护者，让他上传适当的Maven源码至中央Maven仓库。

14.9.4. 打开项目页面

一个Maven POM包含一些开发者可能需要查阅的有很有价值的URL。它们包括项目的web页面，源代码仓库的URL，如Hudson之类的持续集成系统的URL，问题追踪系统的URL。如果这些URL在项目的POM中存在，m2eclipse就能在浏览器中打开这些项目页面。

14.9.5. 解析依赖

你可以配置项目让它从workspace中解析依赖。这种配置改变了Maven定位依赖构件的方式。如果项目被配置成从workspace解析依赖构件，这些构件就不需要存在于你的本地仓库。假设项目a和项目b都在同一个Eclipse workspace中，项目a依赖于项目b。如果workspace依赖解析被关闭了，项目a的Maven构建只有在项目b的构件存在于本地仓库时才会成功。如果workspace依赖解析开启了，m2eclipse就通过eclipse workspace解析这个依赖。换句话说，当workspace依赖解析开启的时候，项目之间的相互关联不需要通过本地仓库安装。

你也可以关闭依赖管理。这种配置的效果是告诉m2eclipse停止管理你项目的classpath，也会从你项目中移除Maven依赖classpath容器。如果你这么做了，管理你项目的classpath就全靠你自己了。

14.10. 使用Maven仓库进行工作

m2eclipse也提供了一些工具使得使用Maven仓库变得容易一些。这些工具提供的功能包括：

- 搜索构件
- 搜索Java类
- 为Maven仓库编制索引

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

14.10.1. 搜索 Maven 构件和 Java 类

m2eclipse为Eclipse Navigation菜单添加几个项目，使搜索Maven构件和Java类变得容易。点击Navigate菜单就能使用这些选项，如图 14.21 “搜索构件和类”:

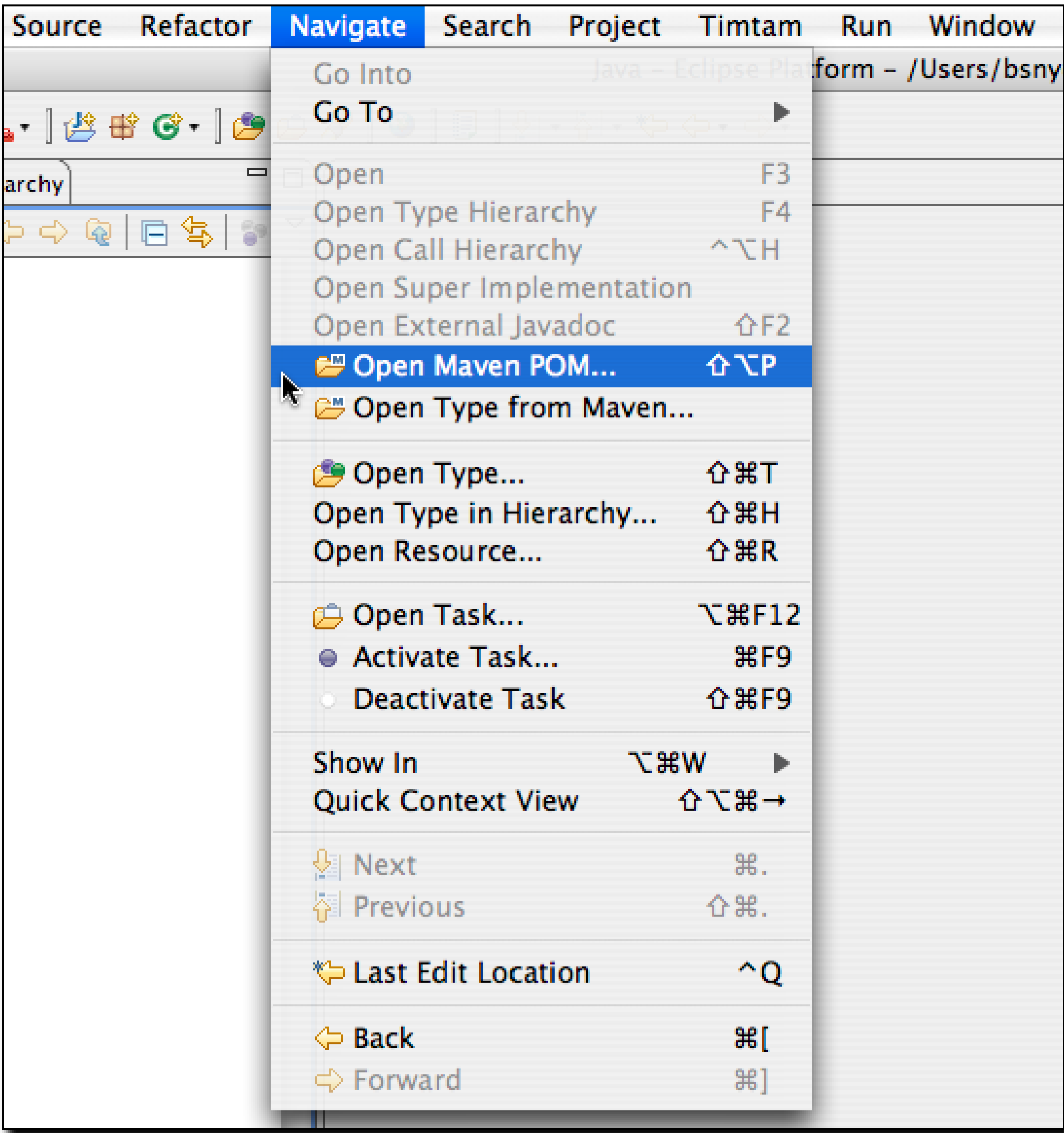


图 14.21. 搜索构件和类

注意在图 14.21 “搜索构件和类”中在Eclipse Navigate菜单下面可用的选项名为Open Maven POM和Open Type from Maven。Open Maven POM选项允许你在Maven仓库中搜索POM，如图 14.22 “搜索一个POM”:

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

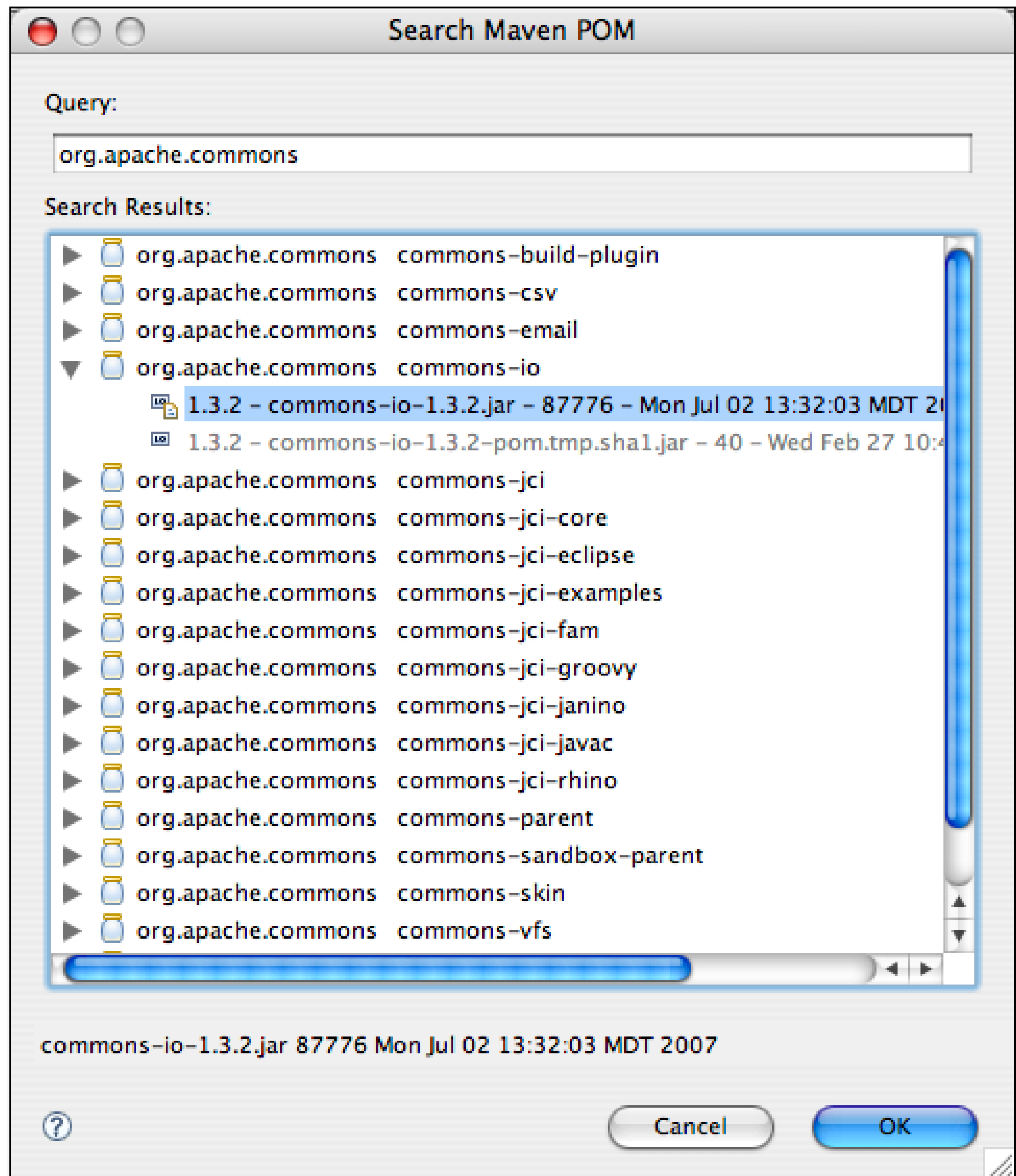


图 14.22. 搜索一个POM

选择一个构件然后点击OK，这个构件的POM在Eclipse被打开以备浏览或者编辑。当你需要快速看一下某个构件的POM的时候，该功能十分方便。

Navigate菜单中第二个m2eclipse选项名为Open Type from Maven。该特性允许你通过名称在远程仓库中搜索一个Java类。打开这个对话框，键入‘factorybean’你就能看到名字带有FactoryBean的很多类，如图 14.23 “在仓库中搜索类”：

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

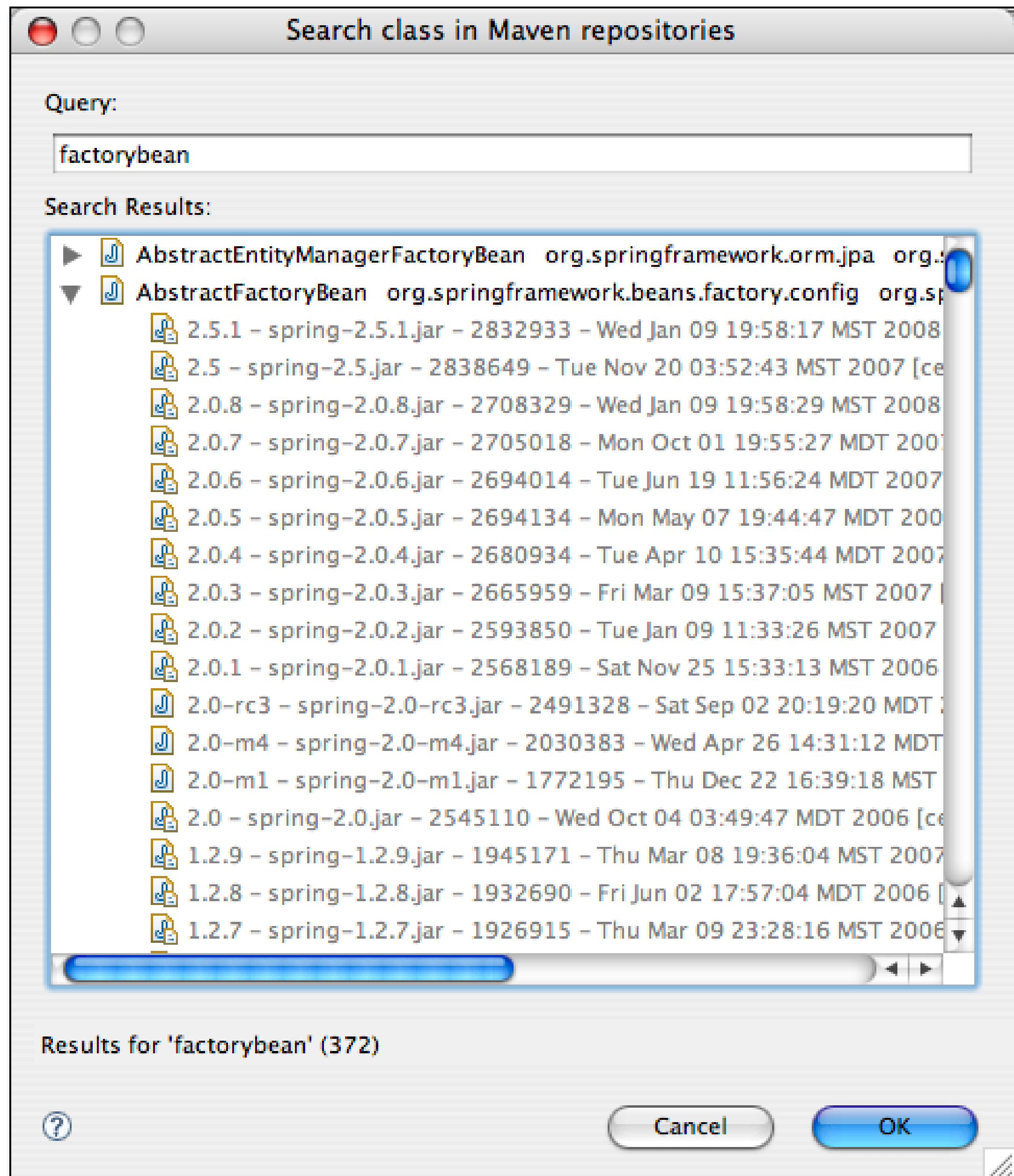


图 14.23. 在仓库中搜索类

这是一个很能节省时间的特性，有了它，手工在Maven仓库中搜索构件中的类成为了过去。如果你需要使用一个特定的类，就打开Eclipse，至菜单Navigate然后搜索类。m2eclipse会显示一个搜索结果构件的列表。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

14.10.2. 为Maven仓库编制索引

Maven索引视图允许你手动的浏览远程仓库的POM并在Eclipse中打开它们。要查看这个视图，打开View → Show View → Other，在搜索框中键入单词“maven”，你应该能看到一个名为Maven索引的视图，如图 14.24 “打开Maven索引视图”：

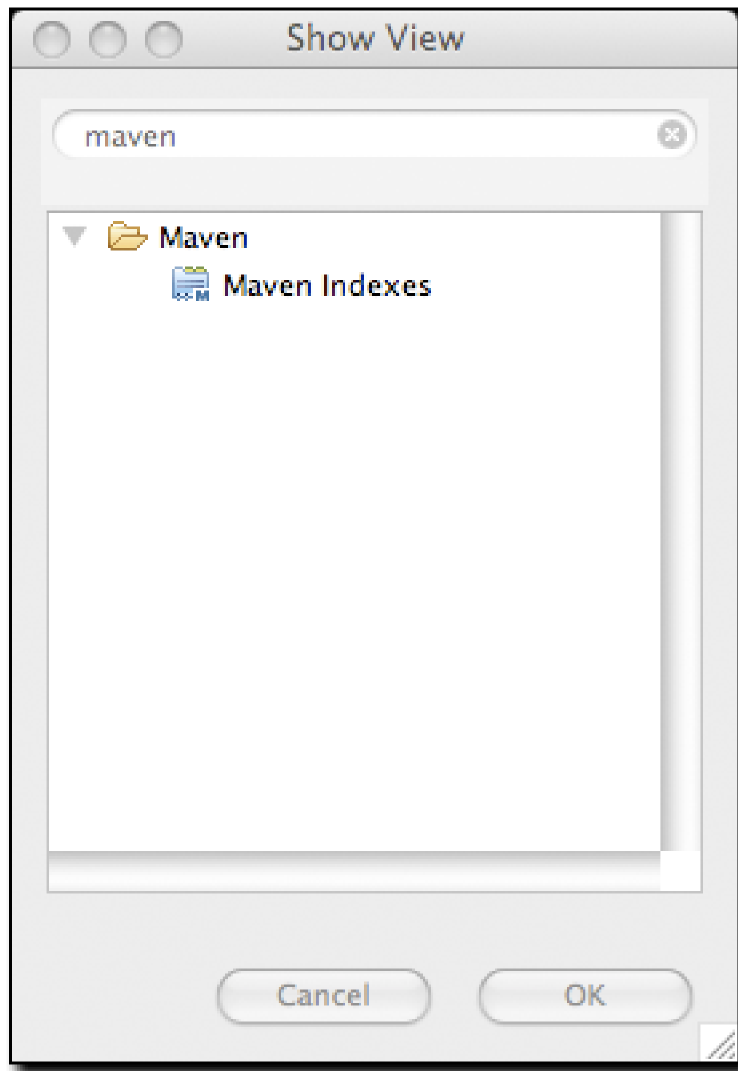


图 14.24. 打开Maven索引视图

选择这个视图然后点击OK。你将会看到如图 14.25 “Maven索引视图”的Maven索引视图。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

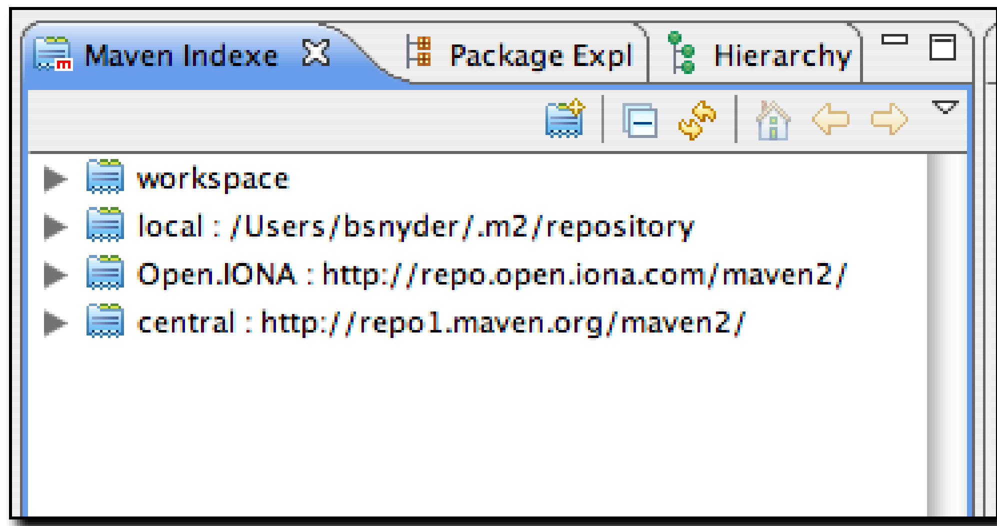


图 14.25. Maven索引视图

此外，图 14.26 “从索引视图定位一个POM”展示手动导航至Maven索引视图之后，定位一个 POM。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

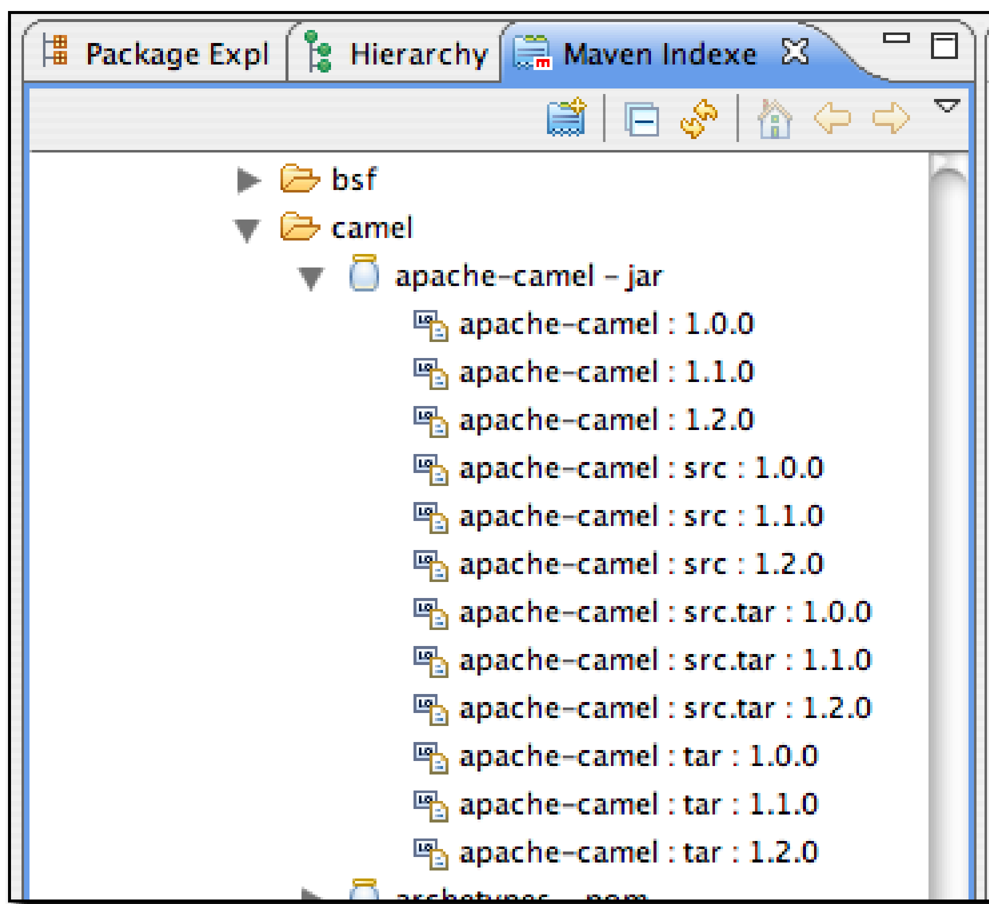


图 14.26. 从索引视图定位一个POM

在找到apache-camel构件之后，双击它会将在Eclipse中打开，以浏览或编辑。

这些特性使得在Eclipse中操作远程仓库变得更快更方便。过去一些年你可能已经花了很多时间来手工的进行这些操作——通过浏览器访问仓库，下载构件然后使用grep程序查找类和POM——你会发现m2eclipse是一种受欢迎的更好的变化。

14.11. 使用基于表单的POM编辑器

m2eclipse的最新版本有个基于表单的POM编辑器，能让你通过一个易用的GUI接口来编辑项目pom.xml的每一个部分。要打开POM编辑器，点击项目的pom.xml文件。如果你为pom.xml文件定制了编辑器，POM编辑器不是默认的编辑器，你可以在这个文件上右击然后选择“Open With... / Maven POM Editor”。POM编辑器会显示Overview标签页如图 14.27 “idiom-core的POM编辑器的Overview标签页”。

一个针对Maven的常见的抱怨是，在十分复杂的多模块项目构件中，它让开发人员面对十分巨大的XML文档。虽然本书的作者相信这只是为类似Maven的工具有带来的弹性所付

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

出的小小的代价，但图形化的POM编辑器这样的工具能让用户在不知道Maven POM背后的XML结构的情况下就能使用Maven。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

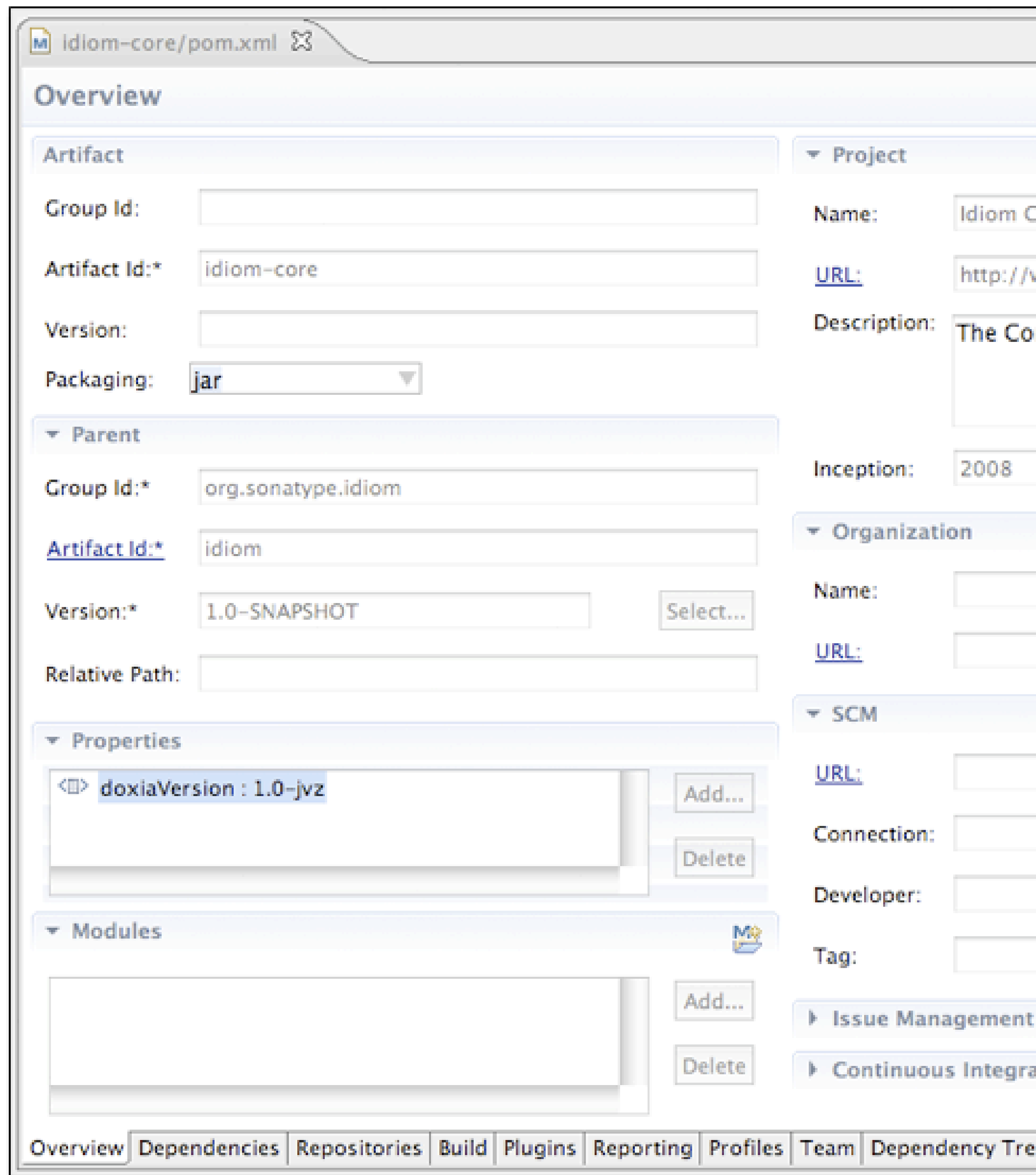


图 14.27. idiom-core的POM编辑器的Overview标签页

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

图 14.27 “idiom-core的POM编辑器的Overview标签页”中展示的项目的artifactId是idiom-core。你会注意到项目idiom-core的大部分字段是空的。POM编辑器中没有groupId或者version，也没有SCM信息。这是因为idiom-core从一个名为idiom的父项目中继承了大部分的信息。如果我们在POM编辑器中打开父项目的pom.xml文件，我们会看到如图 14.28 “idiom父项目的POM编辑器的Overview标签页”的Overview标签页。

整个POM编辑器中各种各样的列表条目上的“打开文件夹”图标说明对应的条目在Eclipse workspace中存在，“jar”图标说明构件引用了Maven仓库。你可以双击这些条目，在POM编辑器中打对应的POM。这对模块，依赖，插件和其它对应于Maven构件的元素都是有效的。POM编辑器中一些部分中带下划线的字段代表了超链接，可以用来为对应的Maven构件打开POM编辑器。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

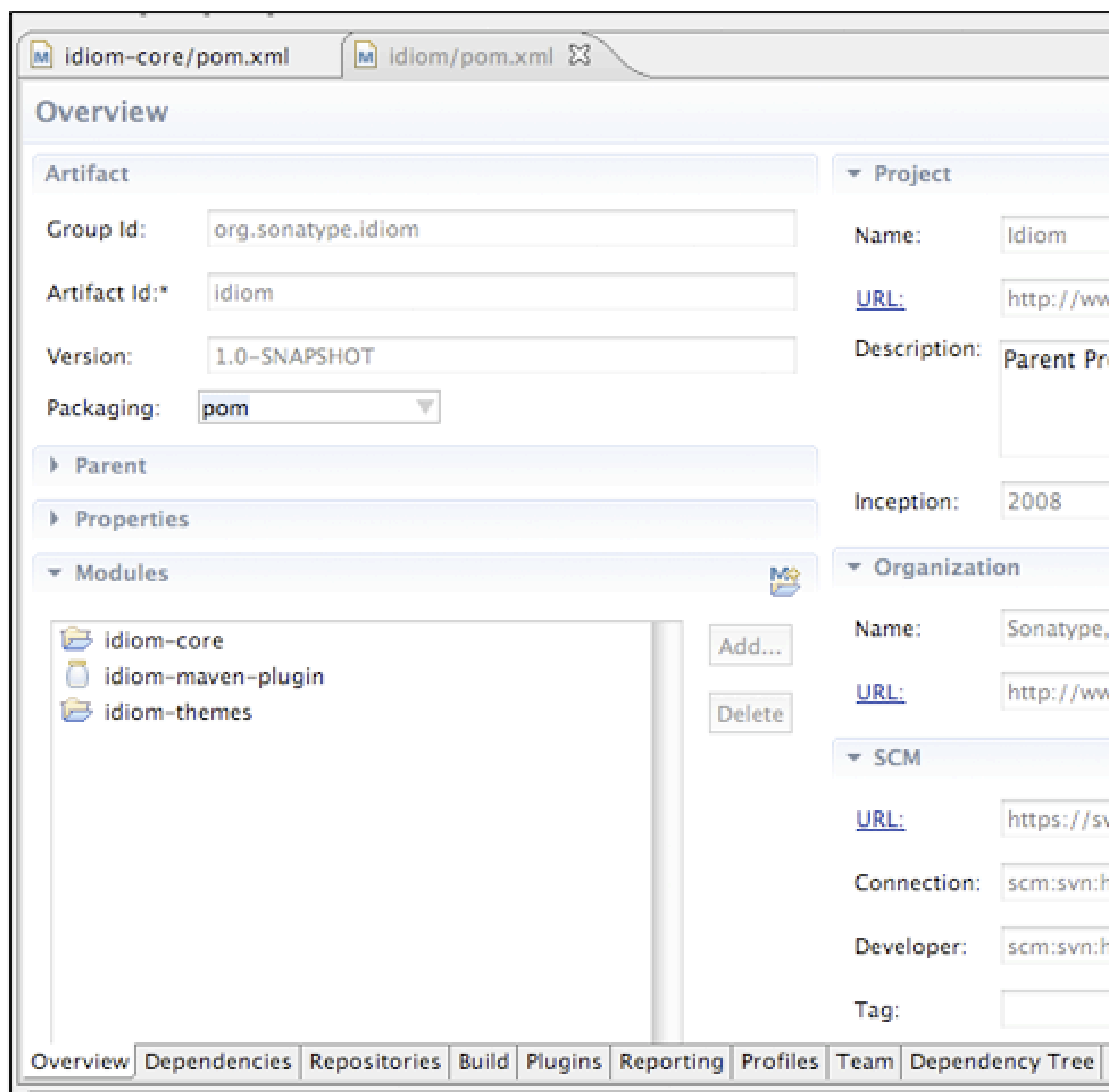


图　14.28.　idiom父项目的POM编辑器的Overview标签页

在这个父POM中，我们看到了groupId和version的定义，它还提供了idiom-core项目所没有很多信息。POM编辑器会给你看你能够编辑的POM内容，它不会给你看任何继承来的值。如果你想在POM编辑器中看idiom-core项目的有效POM，你可以使用POM编辑器右上角工具栏的“Show Effective POM”图标，该图标的样子是蓝色M字母上面有一个左括弧与等于标记。它会在POM编辑器中为项目idiom-code载入有效POM如图　14.29 “idiom-core的有效POM”。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

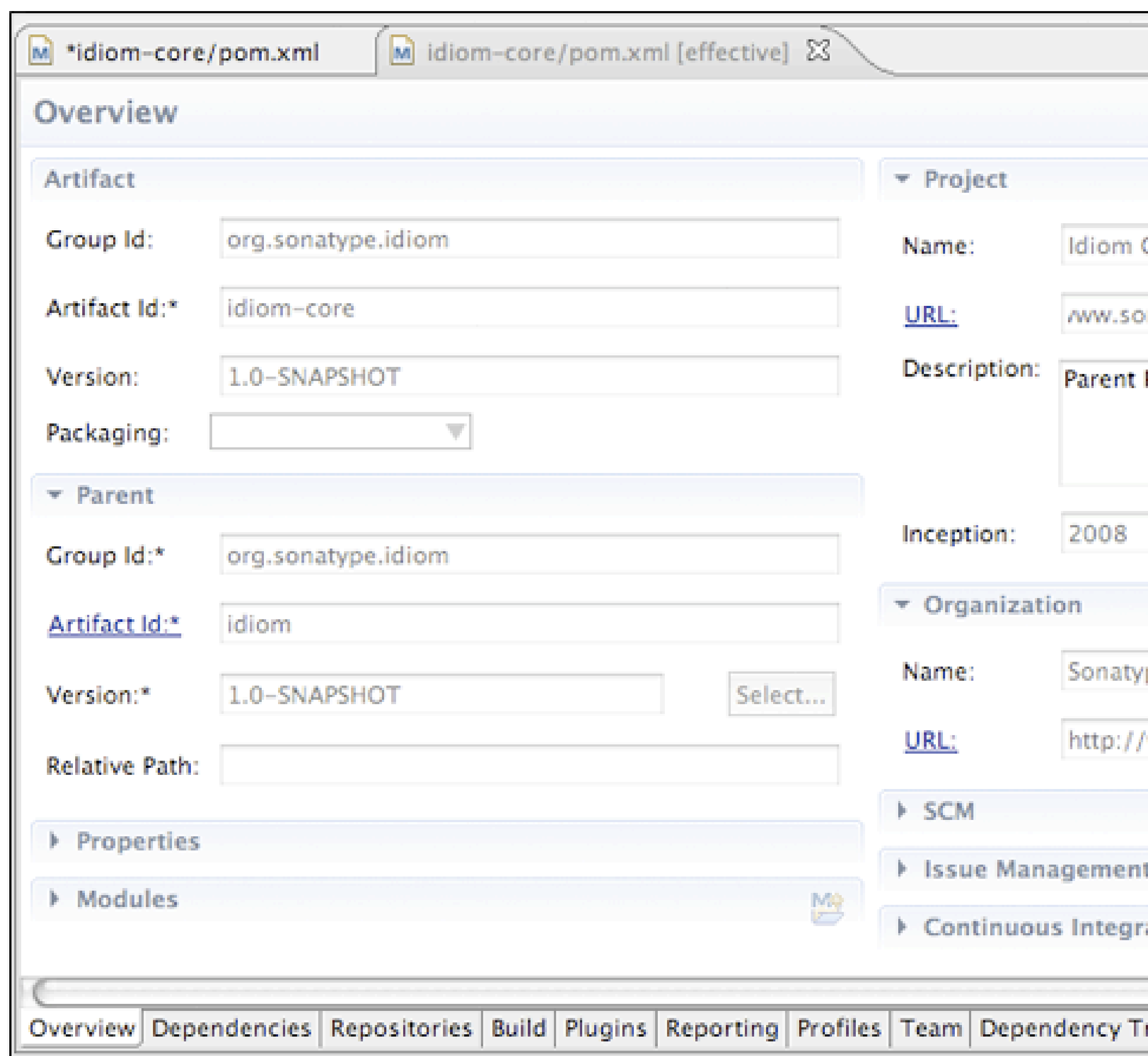


图 14.29. idiom-core的有效POM

这个有效POM归并了`idiom-core`的POM和它祖先的POM（父，祖父，等等。），它其实就是使用了“mvn help:effective-pom”命令为`idiom-core`编辑器显示有效值。由于POM编辑器显示了很多不同POM归并而来的组合视图，因此这个有效POM编辑器是只读的，你不能更新这个有效POM视图中的任何字段。

如果你正在观察图 14.27 “idiom-core的POM编辑器的Overview标签页”中的`idiom-core`项目的POM编辑器，你还能够使用编辑器右上角工具栏上的“Open Parent POM”图标来导航至它的父项目的POM。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

POM编辑器显示了很多来自POM的不同信息。最后一个标签页将pom.xml显示为一个XML文档。在图 14.30 “POM编辑器的Dependencies标签页”中有一个依赖标签页，它暴露了而一个易用的接口以添加和编辑你项目的依赖，以及POM的dependencyManagement部分。m2eclipse插件中的依赖管理屏幕也集成了构件搜索功能。你可以用搜索框来取得“Dependency Details”部分的字段信息。

如果你想知道关于某个构件更多的信息，可以使用“Dependency Details”部分的工具栏的“Open Web Page”图标来查看项目的web页面。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

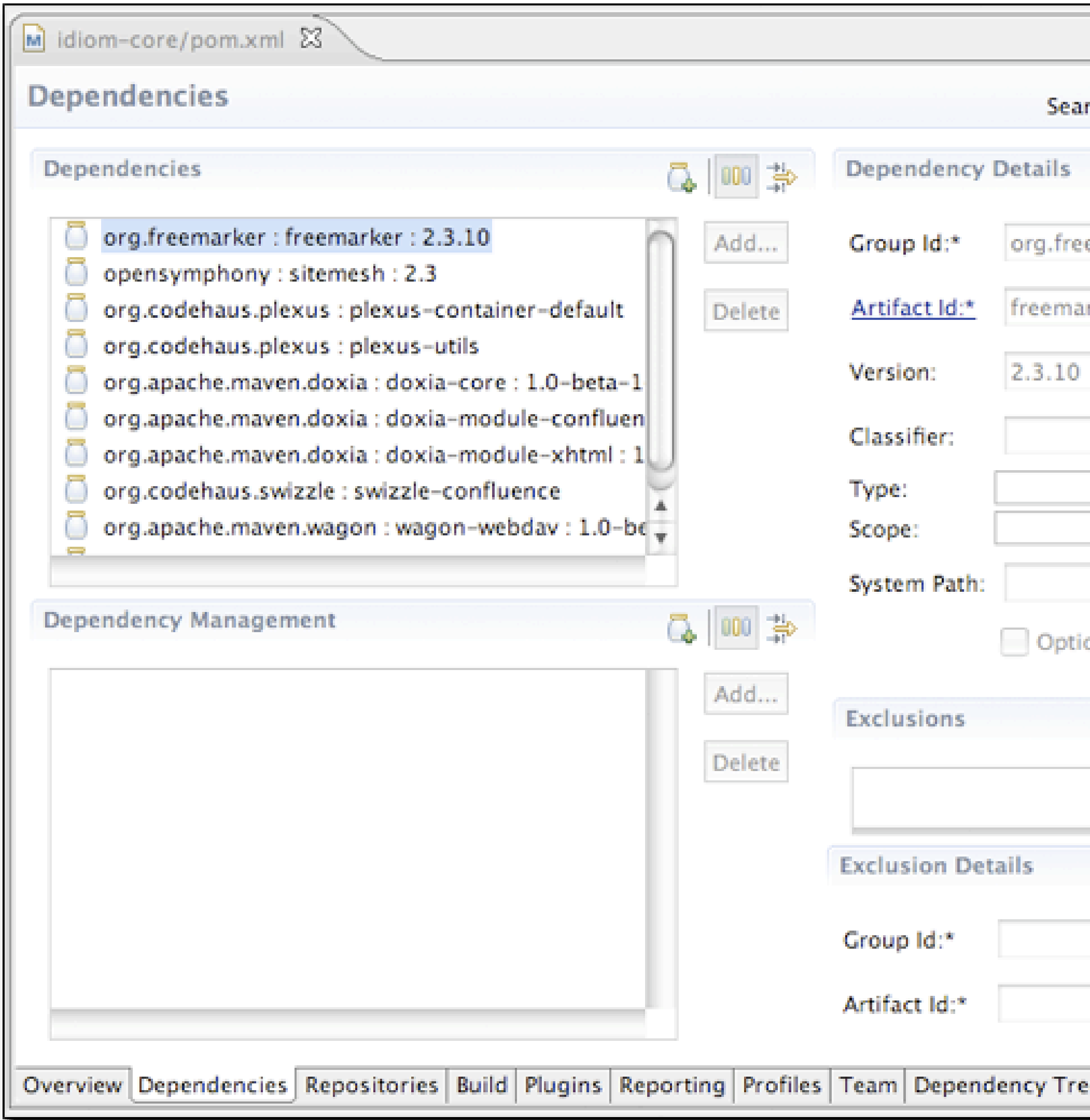


图 14.30. POM编辑器的Dependencies标签页

图 14.31 “POM编辑器的Build标签页”所示的build标签页能让你访问build元素的内容。从这个标签你能够自定义源代码目录，添加扩展，改变默认目标名称，以及添加资源目录。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

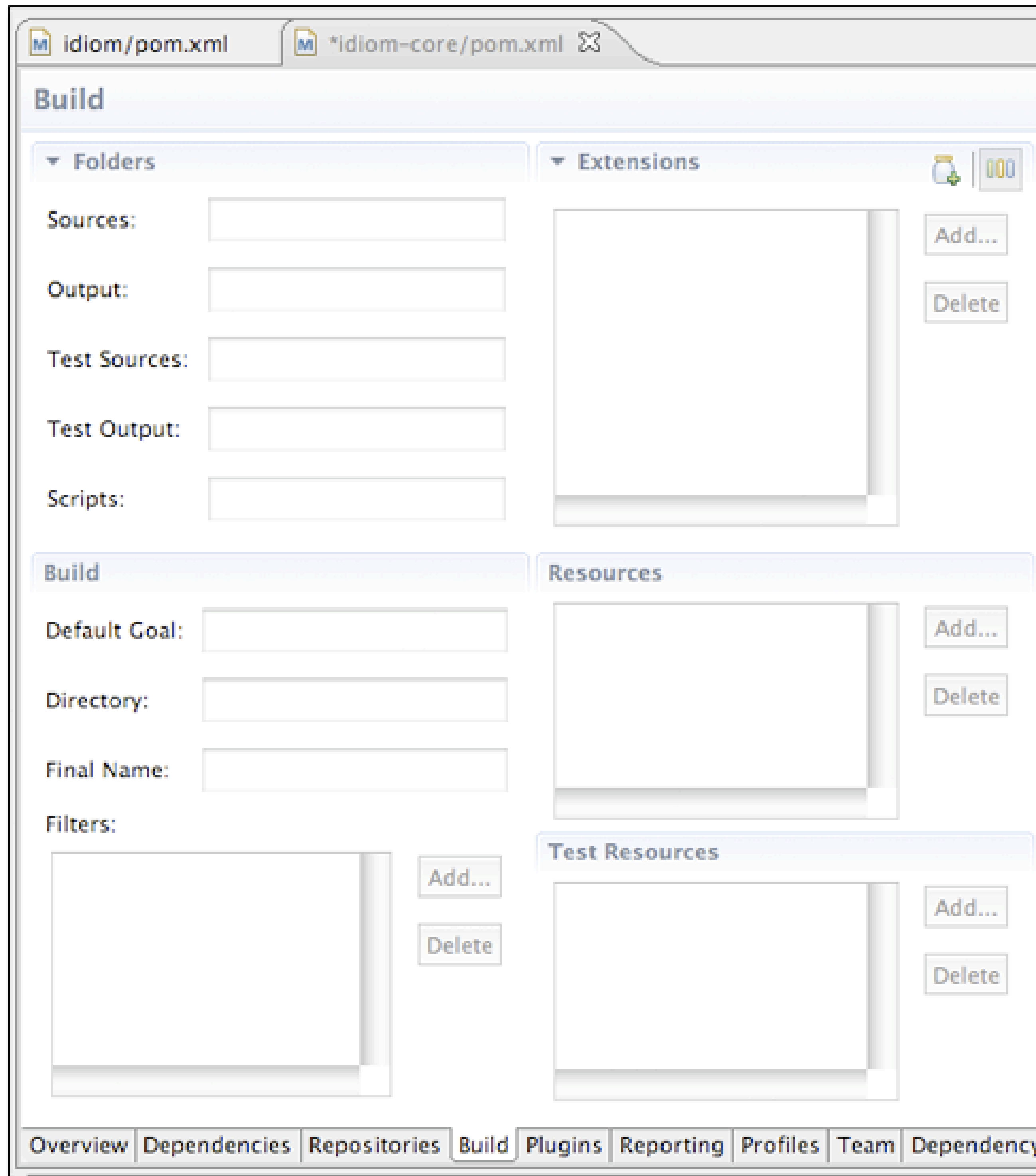


图 14.31. POM编辑器的Build标签页

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

我们只是展示了POM编辑器功能的一个很小的子集。如果你对余下的标签页感兴趣，请下载并安装m2eclipse插件。

14.12. 在m2eclipse中分析项目依赖

最新版本m2eclipse的POM编辑器提供了一些依赖工具。这些工具承诺要改变人们维护及监视项目传递性依赖的方式。Maven的主要吸引力之一是它能够管理项目的依赖。如果你正在编写一个依赖于Spring Framework的Hibernate 3集成的应用程序，你所要做的仅仅是依赖来自中央Maven仓库的spring-hibernate3构件。Maven会读取这个构件的POM然后添加所有必要的传递性依赖。虽然一开始这是一个吸引人们使用Maven的强大特性，但当一个项目有数十个依赖，每个依赖又有数十个传递性依赖的时候这就变得令人费解。

当你依赖于一个项目，这个项目有一个编写得很差的POM，它未能将依赖标记为可选，或者当你开始遇到传递性依赖之间的冲突，这个时候问题就出现了。如果你有一个需求要将类似于commons-logging或servlet-api的依赖排除，又或者你需要弄清楚为什么在某个特定的范围下一个特定的依赖显现了，通常你需要从命令行调用dependency:tree和dependency:resolve目标来追踪那些令人不愉快的传递性依赖。

这个时候m2eclipse的POM编辑器就便捷多了。如果你打开一个有很多依赖的项目，你可以打开Dependency Tree标签页并查看显示为两列的依赖如图 14.32 “POM编辑器的Dependency Tree标签页”。面板的左边显示树状的依赖。树的第一层包含了你项目的直接依赖，每个下一层依赖列出了依赖的依赖。左边的这一块是了解某个特定的依赖如何进入你项目的已解析依赖的很强大方式。面板的右边显示所有已解析的依赖。这是在所有冲突和范围已解决后的有效依赖的列表，也就是你项目将用来编译，测试和打包的有效依赖列表。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

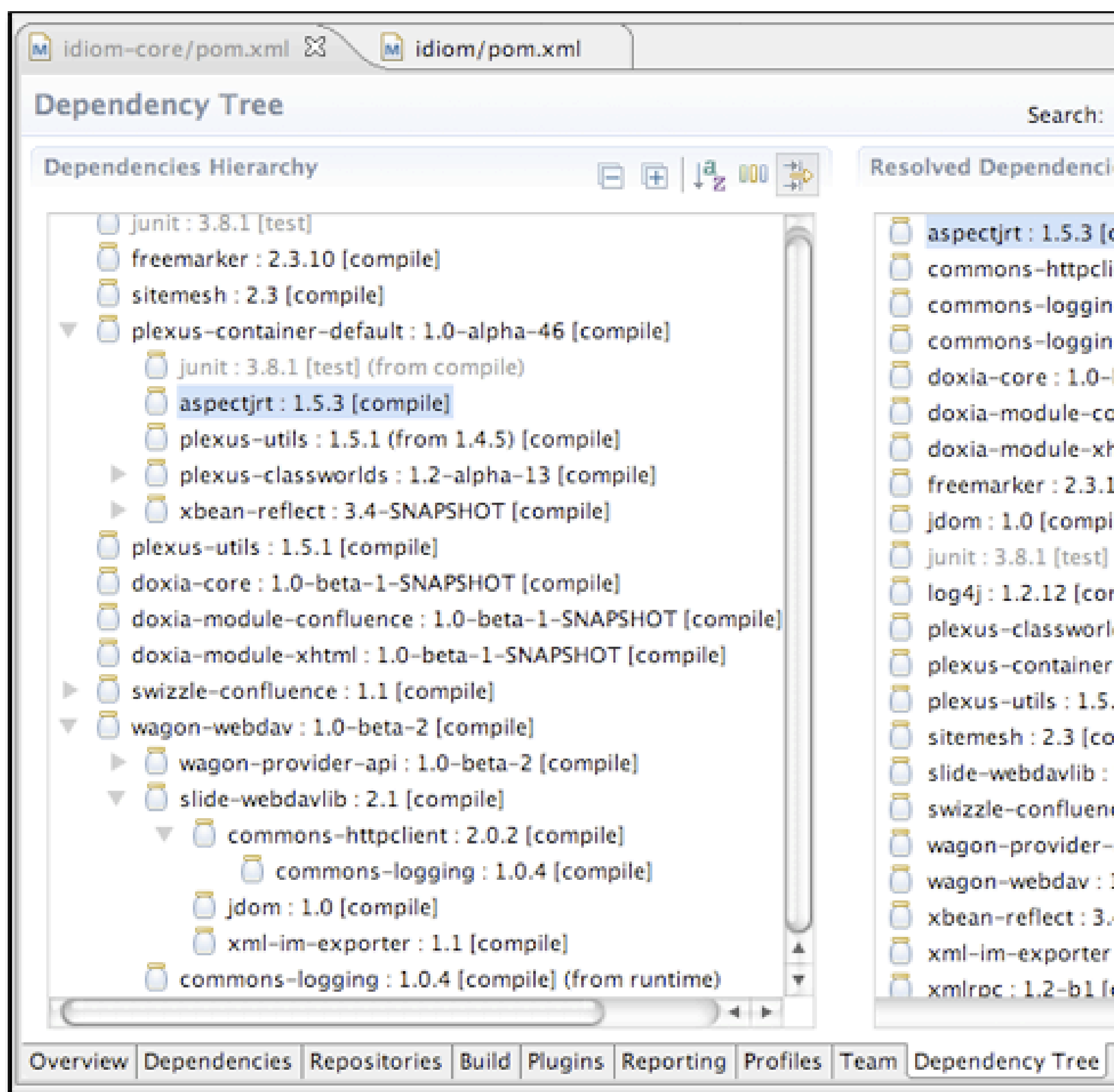


图 14.32. POM编辑器的Dependency Tree标签页

Dependency Tree标签页这一特性非常有价值，因为它能被用作一个侦测工具来找出某个特定的依赖是如何进入已解析的依赖列表的。编辑器中的搜索和过滤功能使得搜索和浏览项目的依赖变得十分容易。你可以使用编辑器工具栏的搜索框和“Dependency Hierarchy”及“Resolved Dependencies”部分的“排序”和“过滤”图标来查看依赖。图 14.33 “在依赖树中定位依赖”展示了当你点击“Resovled Dependencies”列表中的commons-logging的时候会发生什么。当“Dependencies Hierarchy”部分的

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

过滤器被开启的时候，在已解析依赖上点击的时候，面板左边的依赖树会被过滤，以显示所有对已解析依赖起作用的节点。如果你正在试图去除一个已解析依赖，你可以使用这个工具来找出什么依赖（以及什么传递性依赖）对这个已解析的依赖起作用。换句话说，如果你要从你的依赖集合中去除类似于commons-logging这样的依赖，那么Dependency Tree标签页就是你想要使用的工具。

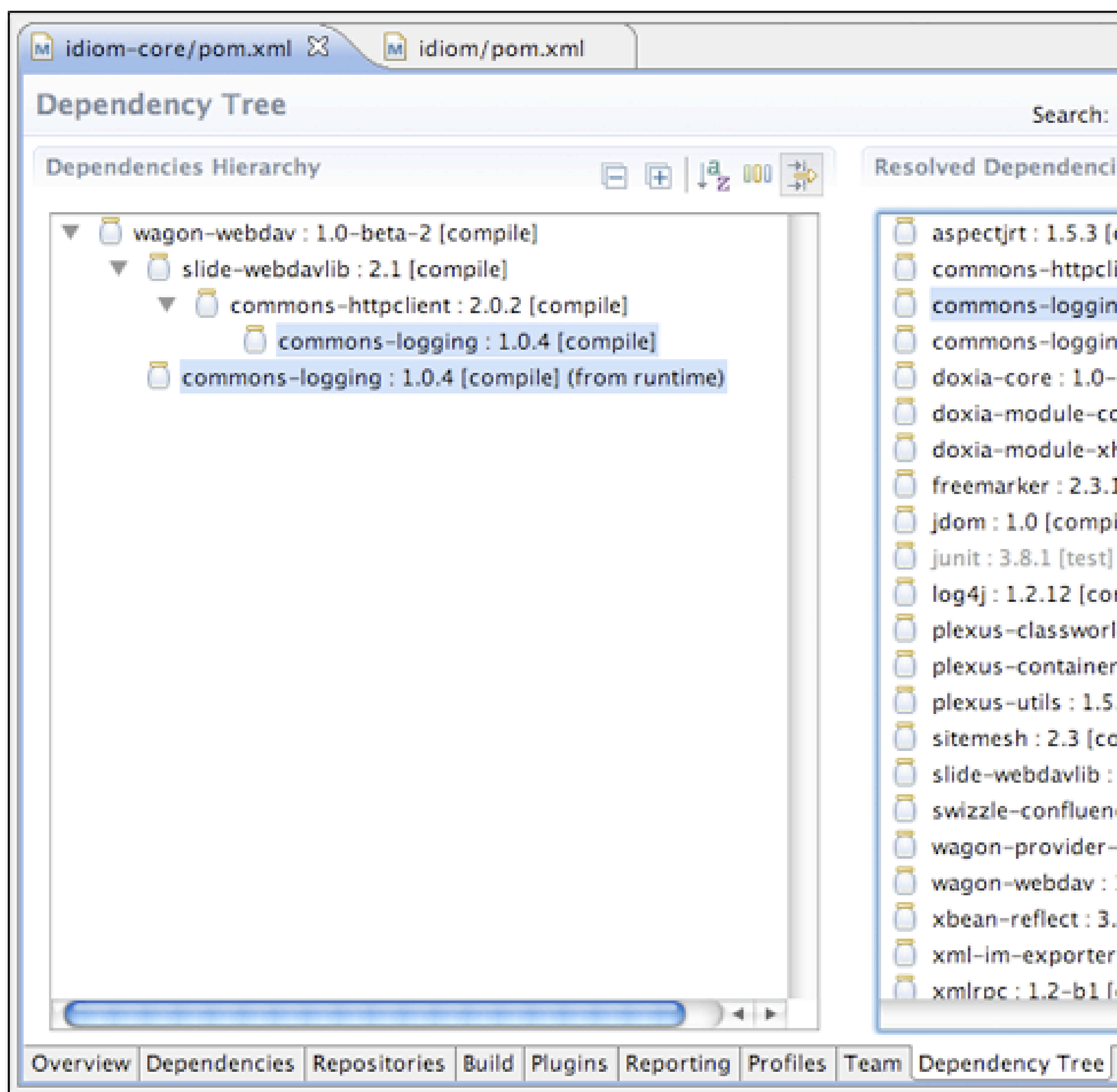


图 14.33. 在依赖树中定位依赖

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

m2eclipse还能让你以图的形式查看你项目的依赖。图 14.34 “以图的形式查看项目的依赖”展示了idiom-core的依赖。最顶上的方框就是idiom-core项目，其它的依赖都在它的下面。直接依赖与顶层方框直接相连，传递性依赖则都最终连接这些直接依赖。你可以在图中选择一个节点，跟它相连的依赖会被标亮，或者你可以使用页面顶部的搜索框来寻找匹配的节点。

注意每个图节点上的“打开文件夹”图标说明这个构件存在与Eclipse workspace中，而“jar”图标说明这个节点的构件指向了Maven仓库。

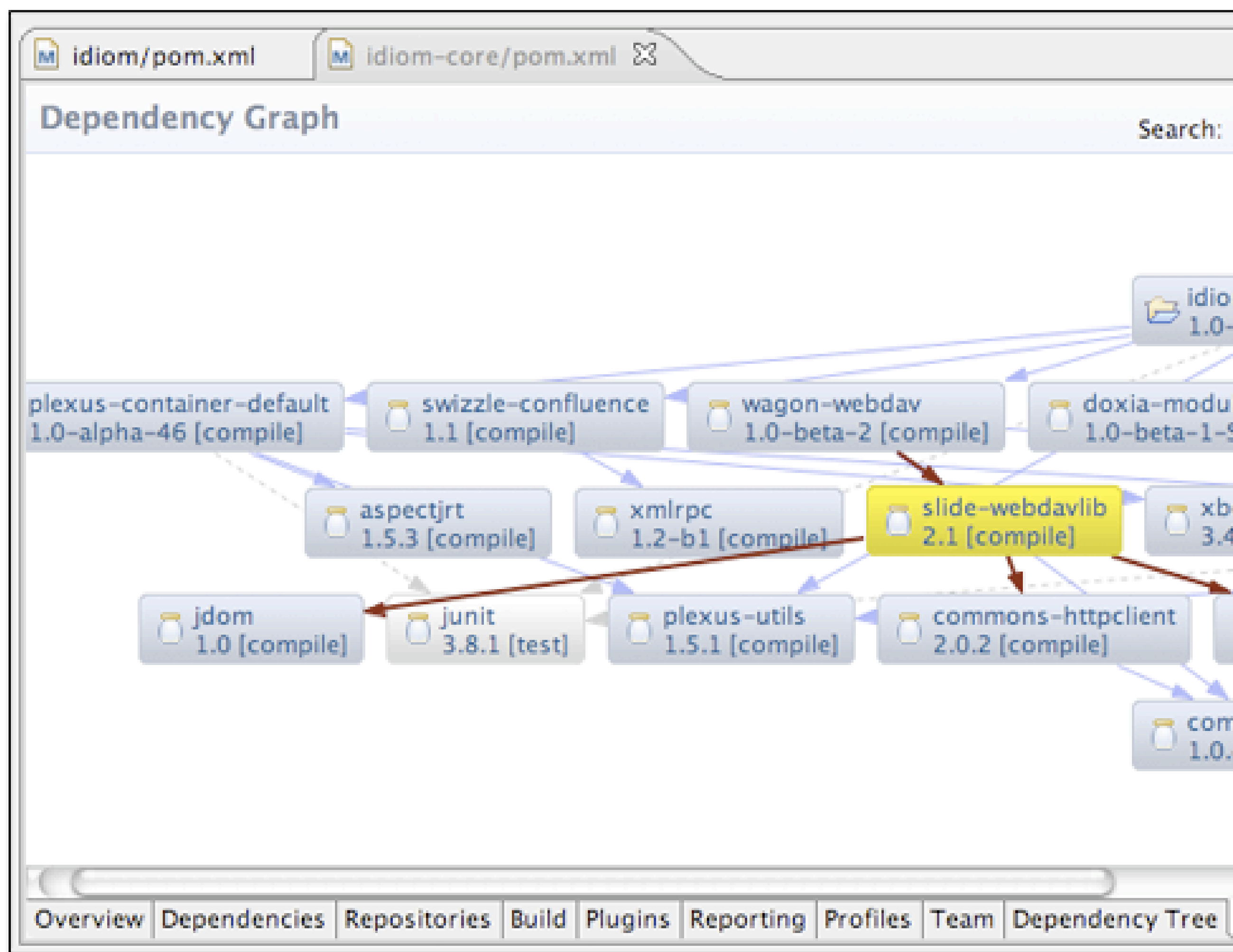


图 14.34. 以图的形式查看项目的依赖

通过在编辑器中右击可以改变图的表现形式。你可以选择显示构件的id，group id，version，scope，或者卷起节点文本，显示图标。图 14.35 “依赖图的放射状布局”展示了和图 14.34 “以图的形式查看项目的依赖”一样的图，但是用了放射状布局。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

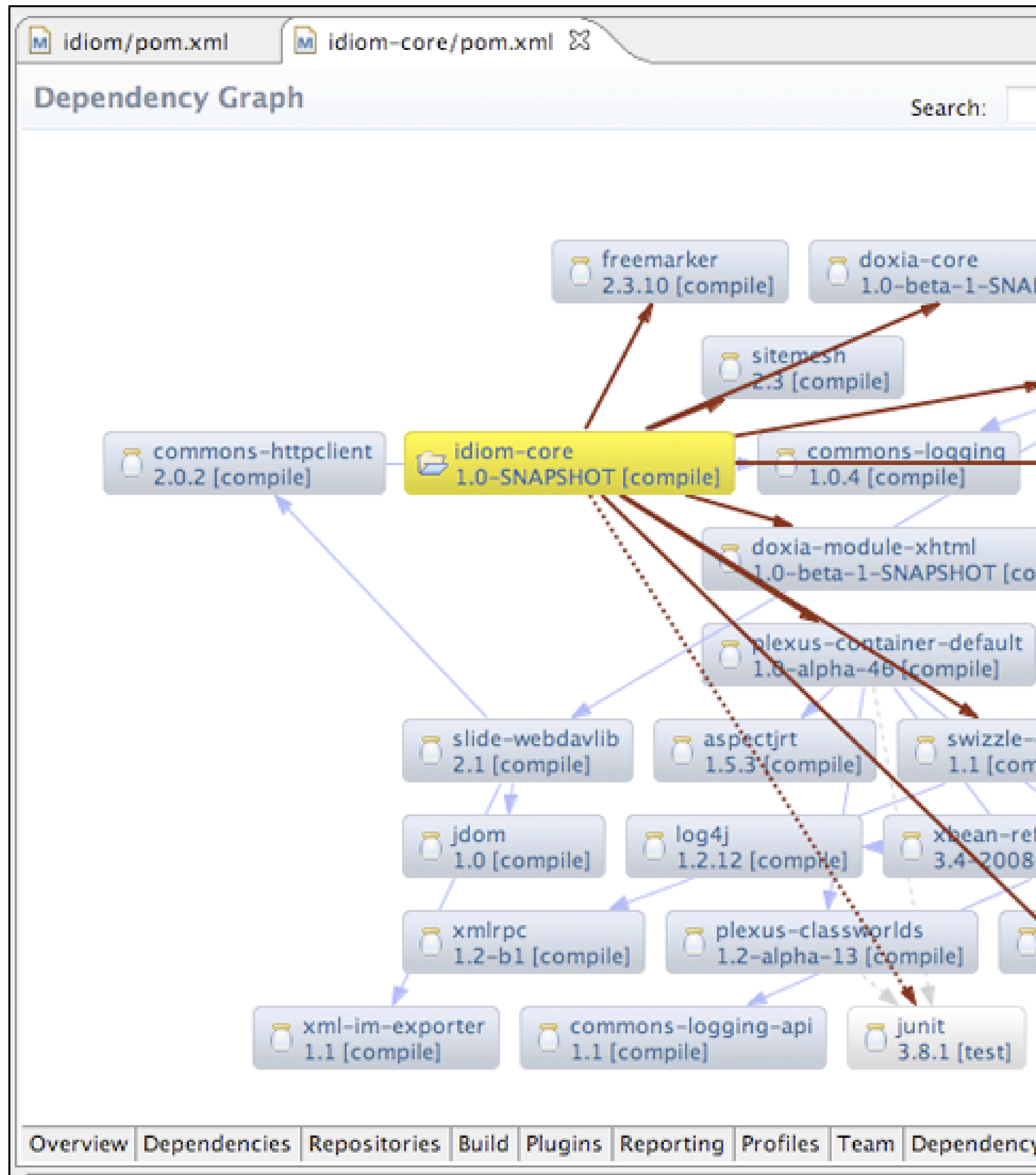


图　14.35.　依赖图的放射状布局

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

14.13. Maven 选项

对于使用Maven开发来说，调整Maven首选项和一些Maven参数是非常重要的，m2eclipse能让你通过Eclipse中的Maven首选项页面来调整一些选项。典型情况下当在命令行使用Maven的时候，这些首选项是可以在目录~/.m2或者命令行选项中设置的。m2eclipse能让你在Eclipse IDE中访问一些最重要的选项。图 14.36 “Eclipse的Maven首选项”展示了在Eclipse中的Maven首选项页面：

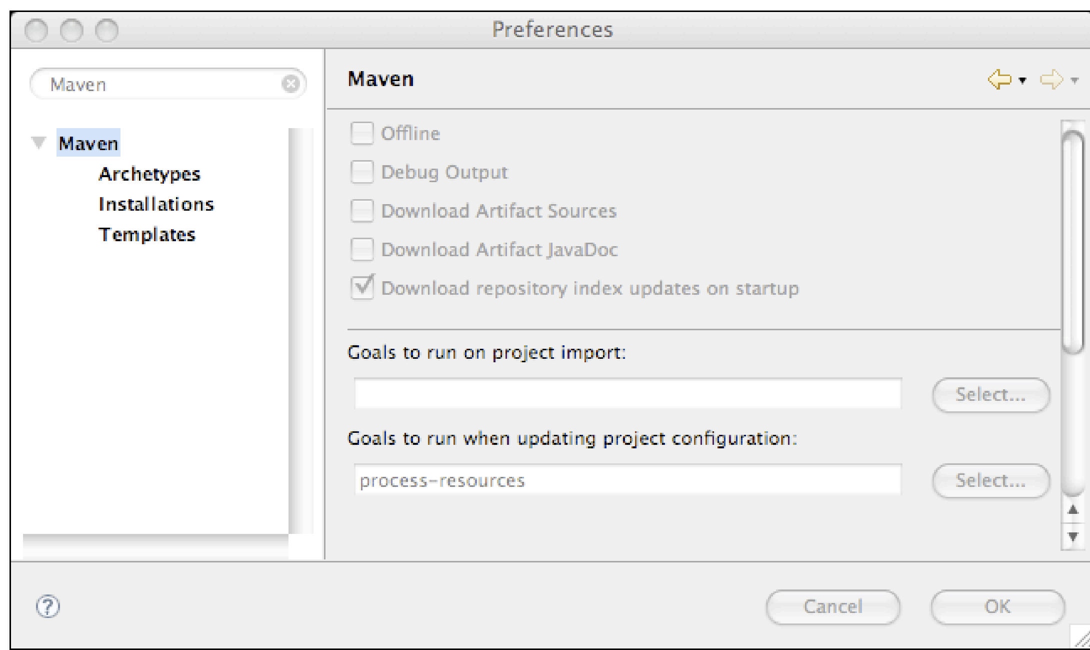


图 14.36. Eclipse的Maven首选项

顶部的复选框提供的功能有：

- 在离线的状态下运行Maven，关闭从远程仓库进行下载的功能。
- 在Maven控制台中开启调试输出
- 从远程Maven仓库下载构件的源码jar文件
- 从远程Maven仓库下载构件的JavaDoc Jar文件
- 在启动的时候下载并更新本地的对远程仓库的索引

下一部分提供了一个弹出式菜单，让你选择当项目被引入以及项目的源码文件夹被更新的时候执行什么Maven目标。默认的目标名为process-resources，它会拷贝并处理项目

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

的资源文件至目标目录，以让项目可以随时打包。如果你需要运行一些自定义的目标以处理资源文件或者生成一些支持性配置，那么定制这个目标列表就非常有用。

如果你需要m2eclipse帮你选择一个目标，点击按钮Select...，会看到“Goals”对话框。左边的图 14.37 “Maven目标对话框”对话框展示了默认Maven生命周期中所有阶段列表的一个目标对话框。

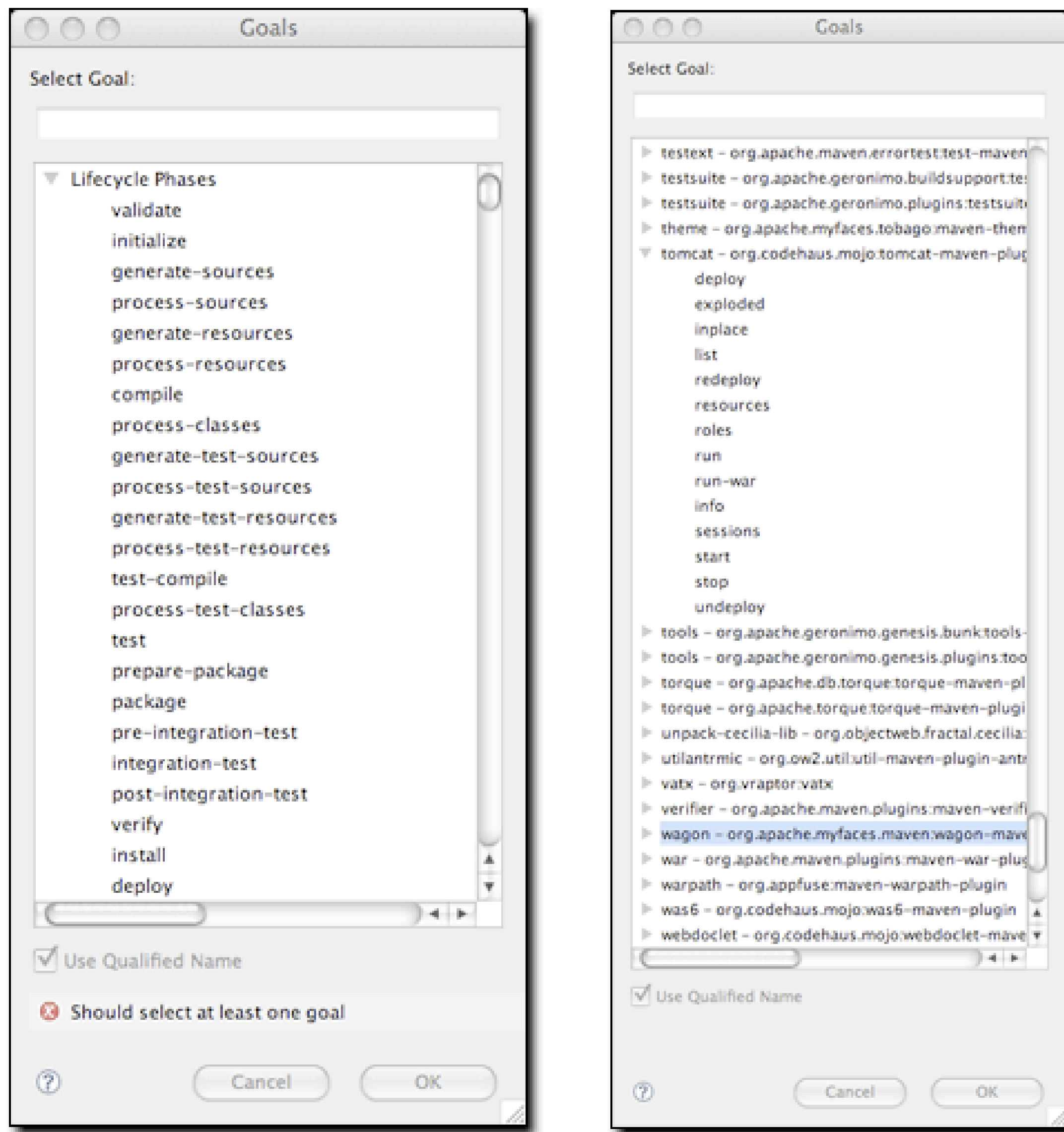


图 14.37. Maven目标对话框

当你第一次看到这个目标对话框的时候，你可能会被它罗列的这么多目标所吓到。事实上，确实有数百个Maven插件做各种各样的事情，从生成数据库，到执行集成测试，到运行静态分析，到使用XFire生成web service。目标对话框中，带有可选的目标的插件

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

有超过200个，右边的对话框图 14.37 “Maven目标对话框”展示了一个Tomcat Maven插件的目标被标亮的“Goals”对话框。你可以通过在搜索框内输入文字来缩小可用目标的数量，当你输入文字的时候，m2eclipse会收缩可用目标的列表，当然，这些目标包含了搜索框内的关键字。

另外一个Maven选项页面是Maven安装配置页面如图 14.38 “Maven安装选项页面”：

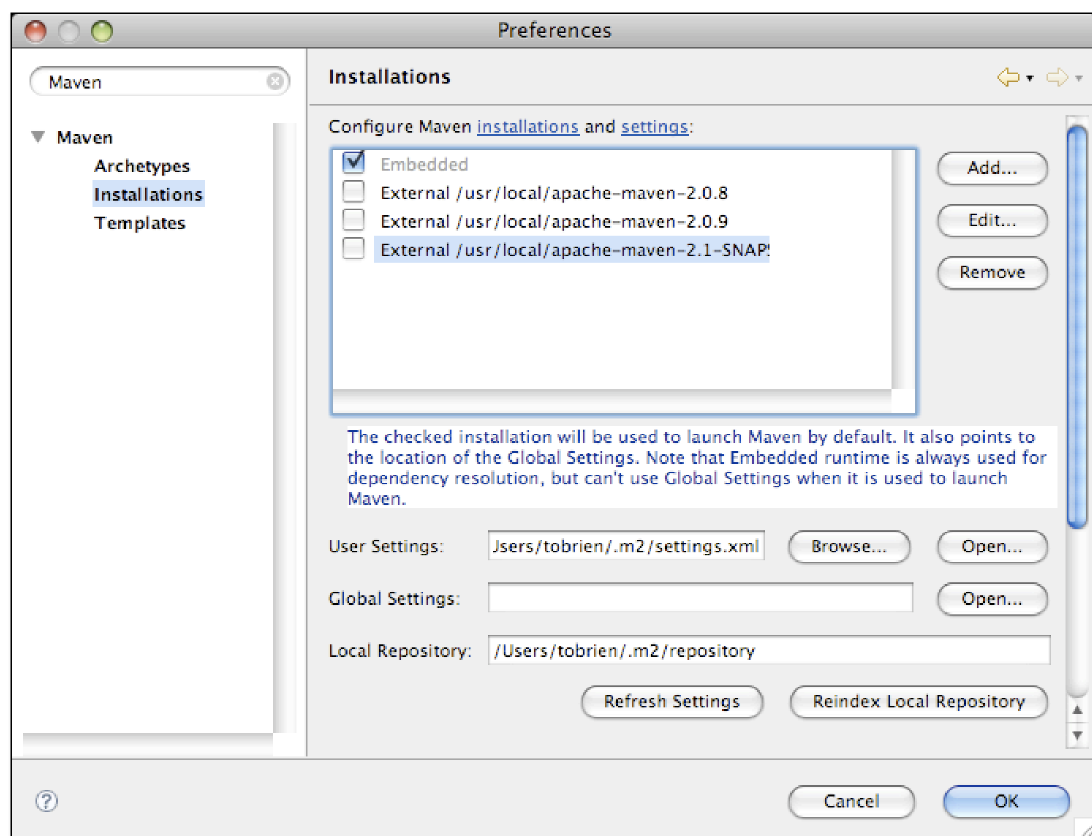


图 14.38. Maven安装选项页面

这个页面能让你往Eclipse环境中添加其它的Maven安装。如果你想要让m2eclipse插件使用一个不同版本的Maven，你可以在这个页面配置多个Maven安装，这非常类似于在Eclipse中配置多个可运行的Java虚拟机。一个被称为Maven嵌入器的Maven的嵌入版本已经被指定了。这就是我们在Eclipse中运行Maven的版本。如果你有另外一个Maven安装，而且你想要用它来运行Maven而不是Maven嵌入器，你可以实时的通过点击Add..来添加另外的Maven。图 14.38 “Maven安装选项页面”展示了一个列出Maven嵌入器，Maven 2.0.9，和Maven 2.1-SNAPSHOT的配置页面。

安装配置页面也允许你指定全局Maven settings文件的位置。如果你不在这个页面指定该文件的位置，Maven会使用位于所选Maven安装目录的conf/settings.xml作为默认全局配置文件。你也可以自定义用户settings文件的位置，默认它位于~/.m2/

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

settings.xml，你还可以自定义你的本地Maven仓库位置，其默认值是~/.m2/repository。

Eclipse选项中还能开启一个装饰器，它的名字是Maven版本装饰器。这个选项可以让你在Eclipse包浏览器中看到项目的当前版本，如图 14.39 “开启Maven版本装饰器”：

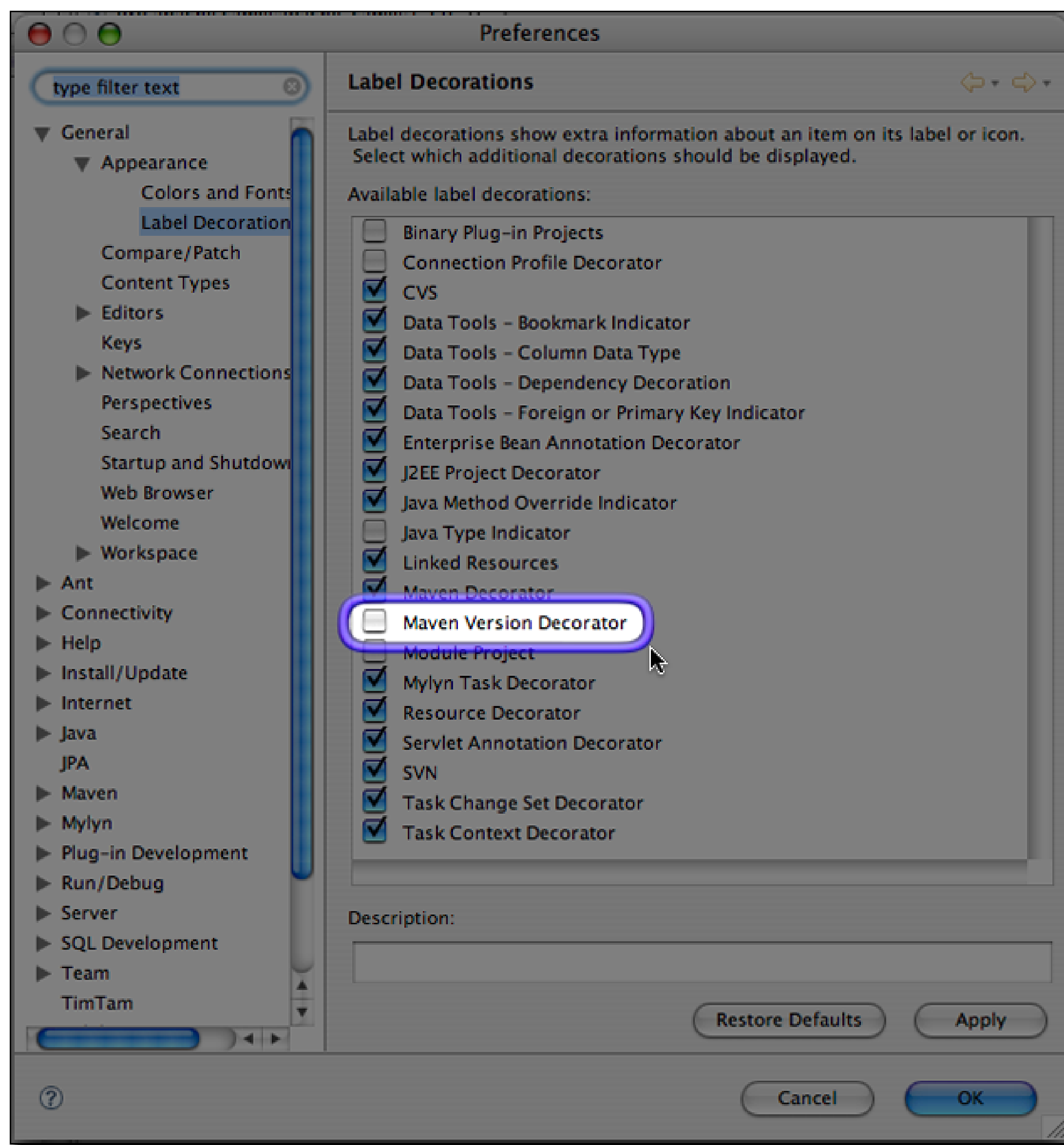


图 14.39. 开启Maven版本装饰器

要开启这个选项，只要选中Maven版本修饰器选项，如在图 14.39 “开启Maven版本装饰器”中标亮的。如果Maven版本修饰器没有开启，项目只会在包浏览器中列出它的名称和相对路径，如图 14.40 “没有Maven版本装饰器的包浏览器”：

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

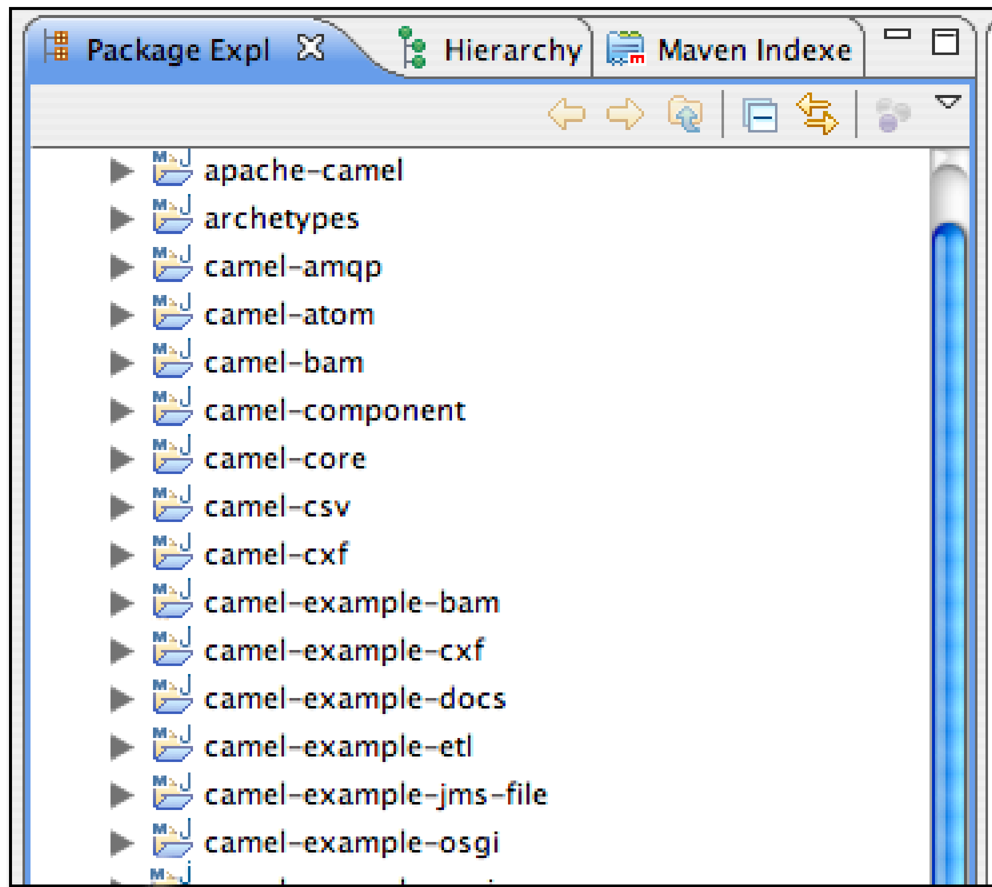


图 14.40. 没有Maven版本装饰器的包浏览器

开启了Maven版本装饰器之后，项目的名称将会包括当前的项目版本，如图 14.41 “开启了Maven版本装饰器的包浏览器”：

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

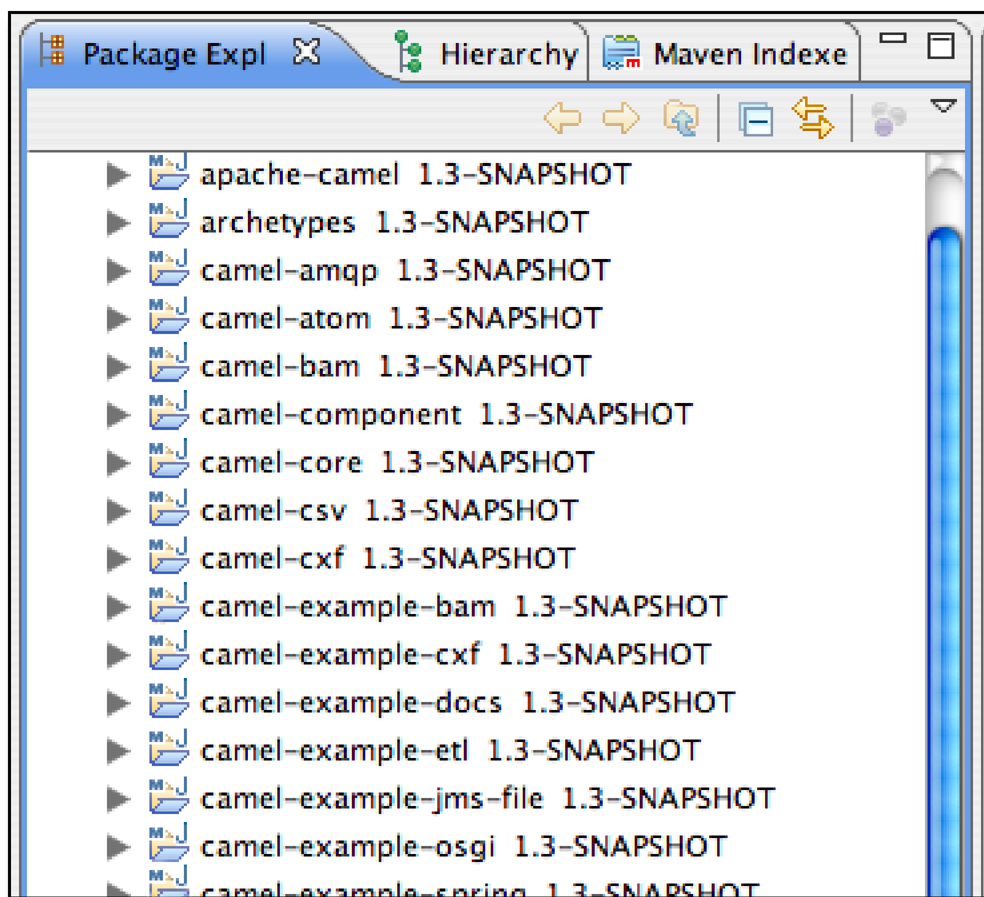


图 14.41. 开启了Maven版本装饰器的包浏览器

这个特性十分有用，你能很方便的看到项目的版本而不再需要打开POM去找version元素。

14.14. 小结

m2eclipse不仅仅是一个为Eclipse添加Maven支持的简单插件，它集成得非常全面，从创建新项目到定位第三方依赖，难易度的降低都是数量级的。m2eclipse是对于一个通晓中央Maven仓库这一丰富语义资源的IDE的第一步尝试。随着越来越多的人开始使用m2eclipse，更多的项目将会发布Maven Archetype，更多的项目会看到往Maven仓库发布源码构件的价值。如果你曾经在没有一个能理解Maven层级结构关系的工具的情况下，尝试一起使用Eclipse和Maven，你就会知道支持嵌套项目的这一特性，对于Eclipse IDE和Maven的集成来说，是至关重要的。

第 15 章 站点生成

15.1. 简介

成功的软件应用程序很少是由单人团队开发的。在讨论任何一个值得我们写下一笔的软件的时候，这个软件背后通常有一个协作的开发者团队，它们大小不一，可能只是一个很少人组成的小型团队，也可能由一个大型分布式环境中数百数千的程序员组成。大部分开源项目（如Maven）成败的原因之一就是它们是否为广泛分布的潜在用户和开发者提供了良好的文档。软件开发主要还是一个协作与交互的实践活动，而发布一个Maven站点是确保你的项目能与最终用户良好沟通的有效手段。

一个开源项目的web站点通常是最终用户和开发人员交流的基础平台。最终用户浏览项目web站点以获得教程，用户指南，API文档，以及邮件列表存档。而开发者浏览项目的web站点获得设计文档，代码报告，问题跟踪（issue tracking），以及发布计划。大型的开源项目可能会和wiki，问题跟踪系统，以及持续集成系统相集成，用那些反映当前开发状态的材料来帮助增强在线文档。如果一个新的开源项目的web站点内容匮乏，不能为预期用户覆盖基本信息，那么这通常预示着这个项目有问题，不会被人采用。对于一个开源项目社区的形成，站点文档和代码本身一样重要。

Maven可被用来创建一个项目web站点，以收集所有与最终用户和开发者相关的信息。不做任何配置，Maven就能生成项目报告，包括单元测试失败，包耦合度，以及代码质量报告。Maven也能让你编写简单的web页面，并基于一致的项目模板呈现这些页面。Maven可以用多种格式发布站点内容，包括XHTML和PDF。Maven可以用来生成API文档，也可以在你项目的二进制发布包内嵌入Javadoc和源代码。在你使用Maven生成了所有最终用户和开发者文档之后，你可以使用Maven将站点发布到远程服务器上。

15.2. 使用Maven构建项目站点

为了展示创建项目站点的过程，使用archetype插件创建一个样例Maven项目：

```
$ mvn archetype:create -DgroupId=org.sonatype.mavenbook -DartifactId=sample-project
```

该命令创建了一个最简单的Maven项目，只带有一个简单的POM，以及`src/main/java`目录下一个Java类。你可以通过简单的运行mvn site构建Maven站点。要在浏览器中预览结果，你可以运行mvn site:run，Maven会构建站点并启动一个内嵌的Jetty容器。

```
$ cd sample-project
$ mvn site:run
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'site'.
[INFO] ------------------------------------------------------------------------
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛


```
[INFO] Building sample-project
[INFO]    task-segment: [site:run] (aggregator-style)
[INFO] ------------------------------------------------------------------------
[INFO] Setting property: classpath.resource.loader.class =>
       'org.codehaus.plexus.velocity.ContextClassLoaderResourceLoader'.
[INFO] Setting property: velocimacro.messages.on => 'false'.
[INFO] Setting property: resource.loader => 'classpath'.
[INFO] Setting property: resource.manager.logwhenfound => 'false'.
[INFO] [site:run]
2008-04-26 11:52:26.981::INFO:  Logging to STDERR via org.mortbay.log.StdErrLog
[INFO] Starting Jetty on http://localhost:8080/
2008-04-26 11:52:26.046::INFO:  jetty-6.1.5
2008-04-26 11:52:26.156::INFO:  NO JSP Support for /, did not find
         org.apache.jasper.servlet.JspServlet
2008-04-26 11:52:26.244::INFO:  Started SelectChannelConnector@0.0.0.0:8080
```

一旦Jetty启动并开始监听8080端口，你就可以通过在浏览器中输入http://localhost:8080/查看项目站点了。你可以在图 15.1 “简单生成的Maven站点”中看到结果。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛



图 15.1. 简单生成的Maven站点

如果你在这个简单的站点中点击查看，你会发现对于一个真实的项目站点来说它并不是很有帮助。这里几乎没什么东西（而且看起来也不怎么样）。由于sample-project并没有配置任何开发者，邮件列表，问题跟踪系统，或者源码仓库，该项目站点的所有这些页面都没有信息。即使是站点状态的首页，“当前没有关于该项目的描述”。要自定义该站点，你需要给该项目及项目POM添加内容。

如果你使用Maven Site插件来构建你的项目站点，你会想要做一些自定义配置。你会想填充POM中的一些重要字段来告诉Maven人们是如何参与该项目的，你也会想要自定义左边的导航菜单，以及页面顶部可见的连接。要自定义站点内容和左边导航菜单的内容，你就需要编辑站点描述符。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

15.3. 自定义站点描述符

当你为站点添加内容的时候，你会想要修改站点左边的导航菜单。以下的站点描述符定制了站点左上角的logo。除了站点的顶部，该描述符还为左边的导航菜单添加了一个菜单小节"Sample Project"。该菜单包含了一个指向概述页面的链接。

例 15.1. 一个初始的站点描述符

```
<project name="Sample Project">
  <bannerLeft>
    <name>Sonatype</name>
    <src>images/logo.png</src>
    <href>http://www.sonatype.com</href>
  </bannerLeft>
  <body>
    <menu name="Sample Project">
      <item name="Overview" href="index.html"/>
    </menu>
    <menu ref="reports"/>
  </body>
</project>
```

该站点描述符引用了一个图片。该图片`logo.png`应该放在`/usr/local/hudson/hudson-home/jobs/maven-guide-zh-to-production/workspace/content-zh/src/site/resources/images`目录下。除了更改站点描述符，你还会想要在`/usr/local/hudson/hudson-home/jobs/maven-guide-zh-to-production/workspace/content-zh/src/site/apt`目录下创建一个简单的`index.apt`页面。将以下的内容写入`index.apt`，它会被转化成`index.html`，当用户访问Maven生成的站点时，他们就会看到这第一个页面。

```
  Welcome to the Sample Project, we hope you enjoy your time
  on this project site.  We've tried to assemble some
  great user documentation and developer information, and
  we're really excited that you've taken the time to visit
  this site.

What is Sample Project

  Well, it's easy enough to explain.  This sample project is
  a sample of a project with a Maven-generated site from
  Maven: The Definitive Guide.  A dedicated team of volunteers
  help maintain this sample site, and so on and so forth.
```

要预览该站点，运行mvn clean site，接着mvn site:run：

```
$ mvn clean site
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛


```
$ mvn site:run
```

在这之后，打开浏览器输入http://localhost:8080。你应该看到类似于截屏图 15.2 “定制样例项目的web站点”的页面。

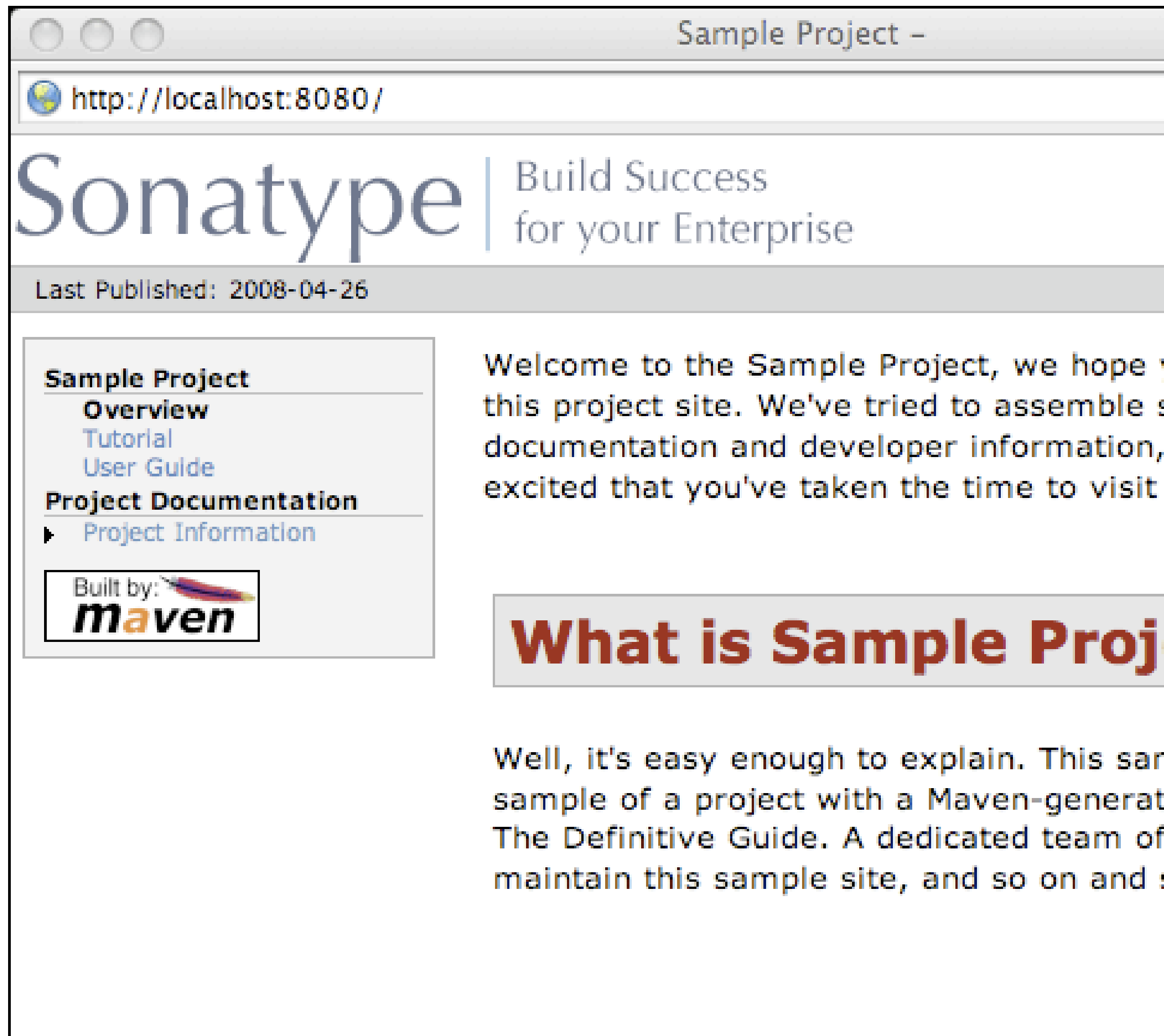


图 15.2. 定制样例项目的web站点

15.3.1. 自定义页面顶端图片

要自定义显示在页面左上角和右上角的图片，你可以使用站点描述符中的bannerLeft和bannerRight元素。

例 15.2. 给站点描述符添加Banner Left和Banner Right

```
<project name="Sample Project">

  <bannerLeft>
    <name>Left Banner</name>
    <src>images/banner-left.png</src>
    <href>http://www.google.com</href>
  </bannerLeft>

  <bannerRight>
    <name>Right Banner</name>
    <src>images/banner-right.png</src>
    <href>http://www.yahoo.com</href>
  </bannerRight>
  ...
</project>
```

bannerLeft和bannerRight元素都有name，src，和href子元素。在上述的站点描述符中，Maven Site插件会生成一个左上角图片为banner-left.png，右上角图片为banner-right.png的站点。Maven会到/usr/local/hudson/hudson-home/jobs/maven-guide-zh-to-production/workspace/content-zh/src/site/resources/images目录寻找这些图片。

15.3.2. 自定义导航菜单

要自定义导航菜单的内容，使用menu元素及其item子元素。menu元素为左边的导航菜单添加一个小节。每个item元素会被渲染成菜单中的一个链接。

例 15.3. 在站点描述符中创建菜单项

```
<project name="Sample Project">
  ...
  <body>

    <menu name="Sample Project">
      <item name="Introduction" href="index.html"/>
      <item name="News" href="news.html"/>
      <item name="Features" href="features.html"/>
      <item name="Installation" href="installation.html"/>
      <item name="Configuration" href="configuration.html"/>
      <item name="FAQ" href="faq.html"/>
    </menu>
    ...
  </body>
</project>
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

菜单项可以嵌套。如果嵌套了菜单项，你就会在导航菜单中创建一个折叠菜单。下述例子添加了一个指向/developer/index.html的"Developer Resources"链接。当用户查看这个Developer Resources页面的时候，其下的菜单项会被展开。

例 15.4. 站点菜单添加链接

```
<project name="Sample Project">
  ...
  <body>
    ...
    <menu name="Sample Project">
      ...
      <item name="Developer Resources" href="/developer/index.html" collapse="true">
        <item name="System Architecture" href="/developer/architecture.html"/>
        <item name="Embedder's Guide" href="/developer/embedding.html"/>
      </item>
    </menu>
    ...
  </body>
</project>
```

当一个菜单项的collapse属性被设置成true的时候，Maven会折叠该菜单项，直至用户浏览了该特定页面。在前面的例子中，如果用户不查看Developer Resources页面，Maven就不会显示System Architecture和Embedder's Guide链接；它只会显示一个指向 Developer Resources链接的箭头。当用户查看Developer Resources页面的时候，其包含的链接会被打开，箭头变成朝下方向。

15.4. 站点目录结构

Maven将所有的站点文档放在src/site目录下。同格式的文档则被放在其子目录下。所有的APT文档应该放到src/site/apt下，所有的FML文档应该被放到src/site/fml下，所有的XDoc文档应该被放到src/site/xdoc下。站点描述符是文件src/site/site.xml，所有的资源应该存储在src/site/resources。Maven Site插件构建web站点的时候，它会从资源目录复制所有文件至站点的根目录。如果你存储了一个文件src/site/resources/images/test.png，那么你就能够在你的站点文档中使用相对路径images/test.png引用该图片。

下面的例子展示了一些文件的位置，包含了APT，FML，HTML，XHTML，和XDoc。注意XHTML内容直接存放在src/site/resources目录。architecture.html文件不会被Doxia处理，它会被直接复制到输出目录，如果你想要包含不被处理的HTML内容，你可以使用这种方式，而不使用Doxia和Maven Site插件的模板和格式化功能。

```
sample-project
+- src/
```

```
  +- site/
     +- apt/
     |  +- index.apt
     |  +- about.apt
     |  |
     |  +- developer/
     |     +- embedding.apt
     |
     +- fml/
     |  +- faq.fml
     |
     +- resources/
     |  +- images/
     |  |  +- banner-left.png
     |  |  +- banner-right.png
     |  |
     |  +- architecture.html
     |  +- jira-roadmap-export-2007-03-26.html
     |
     +- xdoc/
     |  +- xml-example.xml
     |
     +- site.xml
```

注意开发者文档存储在`src/site/apt/developer/embedding.apt`。`apt`目录下的子目录会被反映到站点中最终HTML页面的相对位置上。在Site插件渲染目录`src/site/apt`内容的时候，它会对应的在站点根目录生成HTML输出。如果一个文件在apt目录中，它对应就会生成到web站点的根目录中。如果一个文件在`apt/developer`目录中，它就会被生成到web站点的`developer/`目录下。

15.5. 编写项目文档

Maven使用一个叫做Doxia的文档处理引擎，它读取多个资源格式至一个一般的文档模型。Doxia之后就可以处理文档并渲染结果至不同的输出格式，如PDF或者XHTML。要编写你项目的文档，你需要基于能被Doxia解析的格式编写内容。Doxia现在支持Almost Plain Text (APT), XDoc (一种Maven1的文档格式), XHTML, 和 FML (对FAQ文档很有用) 格式。

本章粗略的介绍一下APT格式。要深入了解APT格式，或者XDoc和FML的深入介绍，请访问如下资源：

- APT参考: http://maven.apache.org/doxia/format.html
- XDoc参考: http://jakarta.apache.org/site/jakarta-site2.html

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

- FML参考: http://maven.apache.org/doxia/references/fml-format.html

15.5.1. APT样例

例 15.5 “APT文档”展示了一个简单的APT文档，它带有一段介绍文字和一个简单列表。注意列表通过伪元素“[]”结束。

例 15.5. APT文档

```
---
Introduction to Sample Project
---
Brian Fox
---
26-Mar-2008
---

Welcome to Sample Project

  This is a sample project, welcome!   We're excited that you've decided to read the
  index page of this Sample Project.  We hope you enjoy the simple sample project
  we've assembled for you.

  Here are some useful links to get you started:

   * {{{news.html}News}}

   * {{{features.html}Features}}

   * {{{faq.html}FAQ}}

   []
```

如果例 15.5 “APT文档”中的APT文档位于`src/site/apt/index.apt`，Maven Site插件就会使用Doxia解析该APT，生成对应于`index.html`的XHTML文档。

15.5.2. FML样例

很多项目维护一个常见问题（FAQ）页面。例 15.6 “FAQ标记语言文档”展示了一个FML文档的例子：

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

例 15.6. FAQ标记语言文档

```
<?xml version="1.0" encoding="UTF-8"?>
<faqs title="Frequently Asked Questions">
  <part id="General">
    <faq id="sample-project-sucks">
      <question>Sample project doesn't work.  Why does sample project suck?</questi
      <answer>
        <p>
          We resent that question.  Sample wasn't designed to work, it was designed
          show you how to use Maven.   Is you really think this project sucks, then
          keep it to yourself.   We're not interested in your pestering questions.
        </p>
      </answer>
    </faq>
    <faq id="sample-project-source">
      <question>I want to put some code in Sample Project, how do I do this?</quest
      <answer>
        <p>
          If you want to add code to this project, just start putting Java source i
          src/main/java.   If you want to put some source code in this FAQ, use the
          source element:
        </p>
        <source>
          for( int i = 0; i < 1234; i++ ) {
            // do something brilliant
          }
        </source>
      </answer>
    </faq>
  </part>
</faqs>
```

15.6. 部署你的项目web站点

一旦项目文档已经写好，并且你已经创建了一个引以为傲的站点，你会想将其部署到服务器上。要部署你的站点，你需要使用Maven Site插件，它会使用很多种方法如FTP，SCP，和DAV，将你的项目站点部署到远程服务器上。要使用DAV部署站点，在POM中配置distributionManagement小节的site元素，如：

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

例 15.7. 配置站点部署

```
<project>
  ...
  <distributionManagement>
    <site>
      <id>sample-project.website</id>
      <url>dav:https://dav.sample.com/sites/sample-project</url>
    </site>
  </distributionManagement>
  ...
</project>
```

distributionManagement中url的值有一个开头标记dav，它告诉Maven Site插件部署该站点至一个理解WebDAV的URL。一旦你已经在我们的sample-project POM中添加了该distributionManagement小节，我们就可以部署该站点：

```
$ mvn clean site-deploy
```

如果你有一个正确配置的，并能理解WebDAV的服务器，Maven就会将项目的web站点部署到远程服务器。如果你正将该项目站点部署到一个公共可见的服务器上，你就会需要配置web服务器的访问证书。比如你的web服务器要求一个用户名和密码（或者其它证书，你可以在你的~/.m2/settings.xml中配置它的值）

15.6.1. 配置服务器认证

在站点部署的过程中配置用户名/密码组合，我们在$HOME/.m2/settings.xml中包含如下的XML：

例 15.8. 在用户特定Settings中存储服务器认证信息

```
<settings>
  ...
  <servers>
    <server>
      <id>sample-project.website</id>
      <username>jdcasey</username>
      <password>b@dp@ssw0rd</password>
    </server>
    ...
  </servers>
  ...
</settings>
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

服务器认证小节可以包含很多认证元素。如果你正使用SCP进行部署，你可能会希望使用公钥认证。为此，不再使用password元素，而是使用publicKey和passphrase元素。根据服务器的配置，可能你仍然需要配置username元素。

15.6.2. 配置文件和目录模式

如果你在一个很大的开发团队中工作，你会想要确保web站点的文件被发布到远程服务器上之后，拥有正确的用户和组权限。要在站点部署过程 中为文件和目录配置特定的模式，在$HOME/.m2/settings.xml中包含如下配置：

例 15.9. 在远程服务器上配置文件和目录模式

```
<settings>
  ...
  <servers>
    ...
    <server>
      <id>hello-world.website</id>
      ...
      <directoryPermissions>0775</directoryPermissions>
      <filePermissions>0664</filePermissions>
    </server>
  </servers>
  ...
</settings>
```

上述设置使所有目录对于所有者和所有者所在用户组可读可写；对于匿名用户则只读，并能列出目录内容。类似的，所有者或所有者所在用户组可以读写任何文件，而其它用户则只有只读访问权限。

15.7. 自定义站点外观

默认的Maven模板可能远不能满足你的期望。如果你想要自定义项目站点，并且不只局限于添加内容，导航元素，和定制logo。Maven提供了很多种方案让你定制web站点，并相继的提供对于内容渲染和站点结构的深入访问。对于小型的，单个项目的站点微调，提供一个定制的site.css通常就足够了。然而，如果你想要自己的自定义信息可以在多个项目中被重用，或者该自定义还涉及了更改Maven生成的XHTML内容，你就应该考虑创建你自己的Maven web站点皮肤。

15.7.1. 自定义站点CSS

最简单的影响项目web站点外观和感觉的方法是使用项目的site.css。就像你为站点提供的任何图片或XHTML内容一样，site.css也被包含于src/site/resources目录

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

中。Maven认为该文件位于src/site/resources/css子目录。使用CSS就可以更改文字风格属性，布局属性，或者甚至添加背景图片和自定义bullet图。例如，如果我们决定让菜单头更显眼一点，就可以在src/site/resources/css/site.css中尝试如下风格。

```
#navcolumn h5 {
  font-size: smaller;
  border: 1px solid #aaaaaa;
  background-color: #bbb;
  margin-top: 7px;
  margin-bottom: 2px;
  padding-top: 2px;
  padding-left: 2px;
  color: #000;
}
```

在你重新生成该站点之后，菜单头应该会有一个灰色背景框，并通过一些额外的边距空间和菜单的其它部分分开。使用该文件，Maven站点的任何结构都可以使用自定义的CSS装饰。如果你在一个特定项目中更改site.css，其变化就至对该项目起作用。如果你想做一些更改并应用到很多个Maven项目中，你可以为Maven Site插件创建一个自定义的皮肤。

提示

对于默认Maven Site模板的结构，没有很好的参考材料。如果你尝试自定义你Maven项目的风格，你应该使用一个Firefox插件如Firebug作为一个工具，来浏览项目页面的DOM

15.7.2. 创建自定义的站点模板

如果默认的Maven站点结构不符合你的要求，你可以自定义Maven站点模板。自定义该模板让你完全控制Maven Site插件的最终输出，也能让你使用非默认的目录结构。

Site插件使用了叫做Doxia的渲染引擎，而它实际上使用Velocity模板将页面渲染成XHTML。要更改默认被渲染的页面结构，我们可以在POM中配置site插件使用自定义的页面模板。site模板相当复杂，你需要有一个关于自定义配置的良好起点。一开始从Doxia的Subversion仓库default-site.vm[1]复制默认的Velocity模板至src/site/site.vm。该模板使用Velocity模板语言编写。Velocity是一个简单的模板语言，它支持简单的宏定义，允许你使用简单标记访问对象的方法和属性。全面的介绍超出了本书的范围，要了解更多的关于Velocity的信息级全面介绍，访问Velocity的项目站点http://velocity.apache.org。

[1] http://svn.apache.org/viewvc/maven/doxia/doxia-sitetools/trunk/doxia-site-renderer/src/main/resources/org/apache/maven/doxia/siterenderer/resources/default-site.vm?revision=595592

default-site.xml模板相当复杂，但自定义左边导航菜单相对的比较简单。如果你试图更改一个menuItem的外观，找到menuItem宏。它位于一个如下的小节中：

```
#macro ( menuItem $item )

  ...

#end
```

如果你使用如下的宏定义替换了默认的宏定义，你将为每个菜单项嵌入Javascript引用，能让用户在不用重新载入整个页面的情况下展开或收缩菜单树。

```
#macro ( menuItem $item $listCount )
  #set ( $collapse = "none" )
  #set ( $currentItemHref = $PathTool.calculateLink( $item.href, $relativePath ) )
  #set ( $currentItemHref = $currentItemHref.replaceAll( "\\", "/" ) )

  #if ( $item && $item.items && $item.items.size() > 0 )
    #if ( $item.collapse == false )
      #set ( $collapse = "collapsed" )
    #else
      ## By default collapsed
      #set ( $collapse = "collapsed" )
    #end

    #set ( $display = false )
    #displayTree( $display $item )

    #if ( $alignedFileName == $currentItemHref || $display )
      #set ( $collapse = "expanded" )
    #end
  #end
  <li class="$collapse">
    #if ( $item.img )
      #if ( ! ( $item.img.toLowerCase().startsWith("http") || $item.img.toLowerCase
        #set ( $src = $PathTool.calculateLink( $item.img, $relativePath ) )
        #set ( $src = $item.img.replaceAll( "\\", "/" ) )
        <img src="$src"/>
      #else
        <img src="$item.img" align="absbottom" style="border-width: 0"/>
      #end
    #end
    #if ( $alignedFileName == $currentItemHref )
      <strong>$item.name</strong>
    #else
      #if ( $item && $item.items && $item.items.size() > 0 )
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛


```
      <a onclick="expand('list$listCount')" style="cursor:pointer">$item.name</a>
      #else
      <a href="$currentItemHref">$item.name</a>
      #end
    #end
  #if ( $item && $item.items && $item.items.size() > 0 )
    #if ( $collapse == "expanded" )
    <ul id="list$listCount" style="display:block">
    #else
    <ul id="list$listCount" style="display:none">
      #end
      #foreach( $subitem in $item.items )
        #set ( $listCounter = $listCounter + 1 )
        #menuItem( $subitem $listCounter )
      #end
    </ul>
  #end
  </li>
#end
```

该更改为menuItem宏添加了一个新的参数。为了使新功能正确工作，你需要更改所有对于该宏的引用，否则最终的模板可能会生成非预期的或者不一致的XHTML。要完成这些引用的更改，在mainMenu宏中进行一次替换。寻找类似下面模板片段的代码，以找到该宏。

```
#macro ( mainMenu $menus )
  ...
#end
```

使用如下的实现替换mainMenu宏。

```
#macro ( mainMenu $menus )
  #set ( $counter = 0 )
  #set ( $listCounter = 0 )
  #foreach( $menu in $menus )
    #if ( $menu.name )
    <h5 onclick="expand('menu$counter')">$menu.name</h5>
    #end
    <ul id="menu$counter" style="display:block">
      #foreach( $item in $menu.items )
        #menuItem( $item $listCounter )
        #set ( $listCounter = $listCounter + 1 )
      #end
    </ul>
    #set ( $counter = $counter + 1 )
  #end
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛


```
#end
```

这个新的mainMenu宏现在和前面的menuItem宏匹配了，同时也为顶层的菜单提供了Javascript支持。点击带有子项的顶层菜单，会得到展开的菜单，能让用户在不用等待页面重新载入的情况下查看整个树。

对于menuItem宏的更改引入了一个expand() Javascript函数。该方法需要被加入到模板文件底部的主XHTML模板中。找到类似于下面的代码片段：

```
<head>
  ...
  <meta http-equiv="Content-Type" content="text/html; charset=${outputEncoding}"
  ...
</head>
```

然后进行如下的替换：

```
<head>
  ...
  <meta http-equiv="Content-Type" content="text/html; charset=${outputEncoding}"
  <script type="text/javascript">
    function expand( item ) {
      var expandIt = document.getElementById( item );
      if( expandIt.style.display == "block" ) {
        expandIt.style.display = "none";
        expandIt.parentNode.className = "collapsed";
      } else {
        expandIt.style.display = "block";
        expandIt.parentNode.className = "expanded";
      }
    }
  </script>
  #if ( $decoration.body.head )
    #foreach( $item in $decoration.body.head.getChildren() )
      #if ( $item.name == "script" )
        $item.toUnescapedString()
      #else
        $item.toString()
      #end
    #end
  #end
</head>
```

在修改了默认站点模板之后，你需要配置项目POM以引用这个新的站点模板。为此，你需要使用Maven Site插件的templateDirectory和template配置属性。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

例 15.10. 在一个项目的POM中自定义页面模板

```
<project>
  ...
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-site-plugin</artifactId>
        <configuration>
          <templateDirectory>src/site</templateDirectory>
          <template>site.vm</template>
        </configuration>
      </plugin>
    </plugins>
  </build>
  ...
</project>
```

现在，你应该能够重新生成你项目的web站点了。这时你可能会注意到maven站点的资源和CSS丢失了。当一个Maven项目自定义站点模板的时候，Site插件认为该项目会提供所有默认的图片和CSS。为了提供项目的资源，你可以从默认的Doxia站点渲染项目中复制资源到你项目的资源目录中，运行如下命令：

```
$ svn co \
        http://svn.apache.org/repos/asf/maven/doxia/doxia-sitetools/trunk/doxia-sit
$ rm \
        doxia-site-renderer/src/main/resources/org/apache/maven/doxia/siterenderer/
        css/maven-theme.css
$ cp -rf \
      doxia-site-renderer/src/main/resources/org/apache/maven/doxia/siterenderer/re
      sample-project/src/site/resources
```

签出doxia-site-renderer项目，删除默认的maven-theme.css文件，然后复制所有的资源到你的src/site/resources目录。

当你重新生成站点的时候，你会注意到一些菜单项看起来像常规未修饰的文本。这是因为站点CSS和新的自定义页面模板发生了一些诡异的交互。修改site.css，为菜单恢复正确的链接颜色后便可修复。只要添加如下配置：

```
li.collapsed, li.expanded, a:link {
  color:#36a;
}
```

在重新生成站点之后，菜单的链接颜色看起来正确了。如果你将这个新的站点模板应用到本章的sample-project项目中，你会看到菜单现在包含了一棵树。点击“Developer

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

Resources”不会再打开“Developer Resources”页面；取而代之的，是展开子菜单。由于你将Developer Resources转换成了一个动态折叠的子菜单，你不再可以通过该菜单打开developer/index.apt页面。为了对付这种变化，你可以在该子菜单下添加一个Overview链接，指向同样的页面：

例 15.11. 给站点描述符添加一个菜单项

```
<project name="Hello World">
  ...
  <menu name="Main Menu">
    ...
    <item name="Developer Resources" collapse="true">
      <item name="Overview" href="/developer/index.html"/>
      <item name="System Architecture" href="/developer/architecture.html"/>
      <item name="Embedder's Guide" href="/developer/embedding.html"/>
    </item>
  </menu>
  ...
</project>
```

15.7.3. 可重用的web站点皮肤

如果你的组织创建了很多Maven项目站点，那么你就会想要在整个组织范围内重用站点模板和定制CSS。如果你想让三十个项目都共享同样的CSS和站点模板，你可以使用Maven对于皮肤的支持。Maven的站点皮肤允许你将资源和模板打包，以让其它项目重用，而不用为所有需要自定义的项目重复创建模板。

你可以定义自己的皮肤，但可能你也会想要使用Maven的可选皮肤。你可以选择下面的几个皮肤。每个皮肤都提供了其自己的导航风格，内容，logo，和模板：

- Maven经典皮肤 - org.apache.maven.skins:maven-classic-skin:1.0
- Maven默认皮肤 - org.apache.maven.skins:maven-default-skin:1.0
- Maven Stylus皮肤 - org.apache.maven.skins:maven-stylus-skin:1.0.1

你可以在Maven仓库中找到一个最新的皮肤完整列表：

http://repo1.maven.org/maven2/org/apache/maven/skins/

创建一个自定义的皮肤其实只是简单的将你自定义的maven-theme.css###封装成一个Maven项目，这样就能根据groupId，artifactId，和version引用它。它可以包含一些资源，如图片，以及用来替换默认模板，生成完全不同XHTML结构的站点模板（用Velocity编写）。大部分情况下，自定义的CSS就可以满足你需要的变化。为了演示该

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

功能，让我们为sample-project项目创建一个皮肤，开始先创建一个自定义的maven-theme.css。

在我们开始编写自定义CSS之前，我们需要创建一个单独的Maven项目，以便让sample-project的站点描述符引用它。首先，使用Maven的archetype插件创建一个基本的项目。在sample-project项目根目录的上一层目录运行如下的命令：

```
$ mvn archetype:create -DartifactId=sample-site-skin -DgroupId=com.sonatyp.maven
```

这会创建一个名为sample-site-skin的项目（以及一个目录）。切换当前目录为sample-site-skin目录，删除所有源码和测试代码，然后创建一个目录来存储皮肤资源：

```
$ cd sample-site-skin
$ rm -rf src/main/java src/test
$ mkdir src/main/resources
```

15.7.4. 创建自定义的主题CSS

接着，为定制皮肤编写自定义的CSS。Maven站点皮肤的CSS文件应该位于src/main/resources/css/maven-theme.css。不像site.css文件位于项目的站点特定源目录那样，maven-theme.css会被打包安装到本地Maven仓库的一个JAR构件中。为了让皮肤JAR文件包含maven-theme.css文件，它需要位于主项目资源目录：src/main/resources。

如同自定义默认站点模板一样，你会想用简单的方法开始定制新的站点CSS。复制默认Maven皮肤的CSS到你的皮肤项目中。要获取默认主题文件的副本，访问maven-default-skin项目的src/main/resources/css/maven-theme.css，将其内容保存到你自己的皮肤文件中。

现在我们拥有了合适的基础主题，使用CSS从这个旧的site.css文件开始定制。使用如下内容替换#navcolumn h5 CSS块：

```
#navcolumn h5 {
  font-size: smaller;
  border: 1px solid #aaaaaa;
  background-color: #bbb;
  margin-top: 7px;
  margin-bottom: 2px;
  padding-top: 2px;
  padding-left: 2px;
  color: #000;
}
```

在成功定制maven-theme.css之后，构建并安装sample-site-skin JAR构件至你的本地仓库，运行：

```
$ mvn clean install
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

在安装完成之后，返回sample-project项目目录，如果你已经在本章前面定制了site.css，将site.css移动为site.css.bak，这样它就不再影响Maven Site插件的输出：

```
$ mv src/site/resources/css/site.css src/site/resources/css/site.css.bak
```

要在sample-project的站点中使用sample-site-skin，你需要在sample-project的站点描述符中添加对于sample-site-skin构件的引用。在站点描述符中使用如下的skin元素来引用皮肤：

例 15.12. 在站点描述符中配置自定义站点皮肤

```
<project name="Sample Project">
  ...
  <skin>
    <groupId>org.sonatype.mavenbook</groupId>
    <artifactId>sample-site-skin</artifactId>
  </skin>
  ...
</project>
```

你可以将Maven站点皮肤想象成一个站点依赖。站点皮肤以构件的形式通过groupId和artifactId被引用。使用站点皮肤允许你在一个单独项目中统一站点定制，并且使得重用自定义CSS和站点模板，像插件重用构建逻辑一样简单。

15.7.5. 在皮肤中自定义站点模板

就像你可以在Maven站点皮肤中自定义CSS一样，你也可以在其中自定义站点模板。Doxia的站点渲染工具会从皮肤JAR文件中找一个META-INF/maven/site.vm文件。为了包含一个自定义的页面模板，复制模板文件至sample-site-skin中正确的位置。将本章前面开发的自定义站点模板复制到sample-site-skin中的src/main/resources/META-INF/maven位置：

```
$ mv sample-project/src/site/site.vm \
    sample-site-skin/src/main/resources/META-INF/maven
```

如果你已经在sample-project中自定义了站点模板，删除Site插件中指向该模板的配置。Site插件会使用站点皮肤中包含的模板来渲染站点。

```
<plugin>
  <artifactId>maven-site-plugin</artifactId>
  <configuration>
    <templateDirectory>src/site</templateDirectory>
    <template>site.vm</template>
  </configuration>
</plugin>
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

Maven站点皮肤应该包含所有它依赖的资源。包括CSS，图片，以及logo。如果在本章前面你已经自定义了站点模板，且已经将默认的doxia-site-renderer资源复制到了src/site/resources目录。你需要将这些文件从sample-project项目移到新的sample-site-skin项目中，使用如下的命令：

```
$ cd ..
$ mkdir -p sample-site-skin/src/main/resources/css
$ mv sample-project/src/site/resources/css/maven-base.css \
    sample-site-skin/src/main/resources/css
$ mkdir -p sample-site-skin/src/main/resources/images
$ mv sample-project/src/site/resources/images/logos \
    sample-site-skin/src/main/resources/images
$ mv sample-project/src/site/resources/images/expanded.gif \
    sample-site-skin/src/main/resources/images
$ mv sample/src/site/resources/images/collapsed.gif \
    sample-site-skin/src/main/resources/images
```

你已经更改了sample-site-skin，因此需要安装该皮肤到本地Maven仓库。在安装好皮肤之后，重新构建sample-project web站点。你会看到新皮肤的自定义站点模板已经应用到了sample-project的web站点上。你会注意到菜单项的颜色可能会有一些不精确，因为你没有为折叠及展开菜单项添加必要的CSS。为此，如下更改src/main/resources/css/maven-theme.css:

```
a:link {
  ...
}
```

至

```
li.collapsed, li.expanded, a:link {
  ...
}
```

重新构建皮肤，然后再次生成站点，你会看到菜单项正常显示了。你已经成功的创建了一个Maven主题，它可以用来将自定义的CSS和模板应用到一组项目中。

15.8. 提示与技巧

本节介绍一些对创建Maven站点很有帮助的提示和技巧。

15.8.1. 给HEAD嵌入XHTML

要给HEAD元素嵌入XHTML，为你项目的站点描述符添加一个head元素。下面的例子为sample-project项目web站点的所有页面添加了一个信息源链接。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

例 15.13. 给HEAD元素嵌入HTML

```
<project name="Hello World">
  ...
  <body>
    <head>
      <link href="http://sample.com/sites/sample-project/feeds/blog"
            type="application/atom+xml"
            id="auto-discovery"
            rel="alternate"
            title="Sample Project Blog" />
    </head>
    ...
  </body>
</project>
```

15.8.2. 在你站点logo下添加链接

如果你工作的项目正由一个组织开发，你会希望能够在项目logo下添加链接。假设你的项目是Apache软件基金会的一部分，你可能想在logo的正下方添加一个指向Apache软件基金会的链接，或者你还想添加一个指向父项目的链接。为了在站点logo下添加链接，直接在站点描述符中的body元素下添加一个links元素。每个links元素下的item元素都会被渲染成logo下的一个链接。下面的例子添加一个指向Apache软件基金会的链接，以及随后一个指向Apache Maven项目的链接。

例 15.14. 在你的站点Logo下添加链接

```
<project name="Hello World">
  ...
  <body>
    ...
    <links>
      <item name="Apache" href="http://www.apache.org"/>
      <item name="Maven" href="http://maven.apache.org"/>
    </links>
    ...
  </body>
</project>
```

15.8.3. 为你的站点添加导航链接

如果你的站点层次结构是一个逻辑层次的一部分，你也许会想要放一系列导航链接，以给用户提供上下文，并让他们能够浏览树状结构中上面的项目，那些项目以子模块的形式包含了当前项目。要配置导航链接，在站点描述符的body元素下添加breadcrumbs元

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

素。每个item元素被渲染成一个链接，并且所有breadcrumbs元素下的item元素会被有序的渲染。导航链接项会从高层到低层进行罗列。下面的站点描述符中，Codehaus项会看起来包含了Mojo项。

例 15.15. 配置站点导航链接

```
<project name="Sample Project">
  ...
  <body>
    ...
    <breadcrumbs>
      <item name="Codehaus" href="http://www.codehaus.org"/>
      <item name="Mojo" href="http://mojo.codehaus.org"/>
    </breadcrumbs>
    ...
  </body>
</project>
```

15.8.4. 添加项目版本

当你为一个有多个版本的项目编写文档的时候，通常在每个页面列出项目的版本号会很有帮助。为了在web站点中显示你项目的版本，只需在站点描述符中添加一个version元素：

例 15.16. 放置版本信息

```
<project name="Sample Project">
  ...
  <version position="left"/>
  ...
</project>
```

这会将版本信息（在sample-project项目中，显示为“Version: 1.0-SNAPSHOT”）放到站点的左上角，在“上次发布”日期的旁边。项目版本可以放置的位置包括：

左边
: 站点Logo下方栏的左边

右边
: 站点Logo下方栏的右边

导航顶部
: 菜单的顶部

导航底部
: 菜单的底部

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

无
: 完全禁止版本

15.8.5. 修改发布日期格式和位置

有一些情况，你可能想要重新格式化或者重新放置项目站点的“上次发布”日期。正如上述的版本提示，你可以使用下列选项之一指定发布日期的位置：

左边
: 站点Logo下方栏的左边

右边
: 站点Logo下方栏的右边

导航顶部
: 菜单的顶部

导航底部
: 菜单的底部

无
: 完全禁止发布日期

例 15.17. 放置发布日期

```
<project name="Sample Project">
  ...
  <publishDate position="navigation-bottom"/>
  ...
</project>
```

缺省情况，发布日期会使用格式`MM/dd/yyyy`进行格式化。你可以更改格式，只需使用`java.text.SimpleDateFormat`的JavaDocs（详细信息参考JavaDoc SimpleDateFormat[2]）中能找到的标准标记。要使用`yyyy-MM-dd`格式化日期，使用如下的`publishDate`元素。

例 15.18. 配置发布日期格式

```
<project name="Sample Project">
  ...
  <publishDate position="navigation-bottom" format="yyyy-MM-dd"/>
  ...
</project>
```

[2] http://java.sun.com/j2se/1.5.0/docs/api/java/text/SimpleDateFormat.html

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

15.8.6. 使用Doxia宏

除了高级的文档渲染特性，Doxia同时也提供了一个宏引擎，它允许每种输入格式触发动态内容的注入。一个优秀的例子是snippet宏，它能让一个文档从一个HTTP可访问的源文件抓取代码片段。使用该宏，一小段APT可以被渲染成XHTML。下面的APT代码调用了snippet宏。请注意该代码应该是单独连续的一行，插入一个反斜杠是为了表示换行，以让代码更适合页面。

```
%{snippet|id=modello-model|url=http://svn.apache.org/repos/asf/maven/archetype/trun
maven-archetype/maven-archetype-model/src/main/mdo/archetype.mdo}
```

例 15.19. XHTML中Snippet宏的输出

```
<div class="source"><pre>

<model>
  <id>archetype</id>
  <name>Archetype</name>
  <description><![CDATA[Maven's model for the archetype descriptor.]]></description>
  <defaults>
    <default>
      <key>package</key>
      <value>org.apache.maven.archetype.model</value>
    </default>
  </defaults>
  <classes>
    <class rootElement="true" xml.tagName="archetype">
      <name>ArchetypeModel</name>
      <description>Describes the assembly layout and packaging.</description>
      <version>1.0.0</version>
      <fields>
        <field>
          <name>id</name>
          <version>1.0.0</version>
          <required>true</required>
          <type>String</type>
        </field>
        ...
      </fields>
    </class>
  </classes>
</model>

</pre></div>
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

警告

在APT源文档中，绝对不要缩进Doxia宏。这么做会让APT解析器完全的忽略该宏。

要了解更多如何在你代码中定义snippet，以及snippet宏的信息，参考Maven站点上Snippet宏的指南：http://maven.apache.org/guides/mini/guide-snippet-macro.html。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

第 16 章 仓库管理器

16.1. 简介

仓库管理器有两个服务目的：首先它的角色是一个高度可配置的介于你的组织与公开Maven仓库之间的代理，其次它为你的组织提供了一个可部署你组织内部生成的构件的地方。

代理Maven仓库有很多好处。对于一开始使用Maven的情况来说，通过为所有的来自中央Maven仓库的构件安装一个本地的缓存，你将加速组织内部的所有构建。如果有开发人员想要下载 Spring Framework 的2.5版本，并且你在使用Nexus，那些依赖（以及依赖的依赖）只需要从远程仓库下载一次。如果有一个高速的 Internet 网络连接，这看起来没什么大不了，但是如果你一直要求你的开发人员去下载几百兆的第三方依赖，那么真正节省的时间将会是Maven检查依赖新版本以及下载依赖的时间。通过本地仓库提供Maven依赖服务可以节省数百的HTTP请求，在大型的多项目构建中，这样可以为一次构件节省几分钟的时间。

除了简单的时间和带宽的节省，仓库管理器为组织提供了一种控制Maven下载的机制。你可以详细的设置从公开仓库包含或排除特定的构件。能够控制从核心Maven仓库的下载对于很多组织来说是经常是一个必要前提，它们需要维护一个组织中使用依赖的严格控制。一个想要标准化某个如Hibernate或者Spring依赖版本的组织可以通过在仓库管理器中仅仅提供一个特殊版本的构件来加强这种标准。还有一些组织可能关心确保所有外部的依赖拥有和组织的法律规范相容的许可证。如果一个企业生产了一个分发应用程序，它们可能想要确定没有人不小心添加了一个涉及GPL许可证的依赖。仓库管理器为那些需要确信总体架构和政策实施的组织提供了这一层次的控制。

除了控制对远程仓库的访问以外，仓库管理器也为Maven的全面使用提供了一些很至关重要的东西。除非你希望你组织的每一个成员下载并构建一个单独的内部项目，否则你会希望为开发人员和部门之间提供一种共享内部项目构件的快照版本和发布版本的机制。Nexus为你的组织提供了这样的部署目标。在你安装了Nexus之后，你可以开始使用Maven让它部署快照版和发布版至一个由Nexus管理的定制仓库。

16.1.1. Nexus历史

Tamas Cservenak 在2005年12月开始为Proximity工作，当时他正想办法将它自己的系统和由Hungarian ISP提供的慢得难以置信的ADSL连接隔离开。Proximity以一个简单web应用的形式启动，用来为有网络连接问题的小型组织代理构件。为Maven构件创建一个对中央核心仓库的本地的命令驱动的缓存，能让组织访问来自中央核心仓库的构件，而且它同时也能确保这些构件不会通过很慢的ADSL连接来下载，要知道很多开发人员在使用这个连接。在2007年，Sonatype请求Tamas帮助创建一个类似的名为Nexus的产品。Nexus目前可以被认为是Proximity逻辑上的下一个步伐。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

Nexus目前有一个活动的开发团队包括Tamas Cservenak，Max Powers，Dmitry Platonoff 和Brian Fox。Nexus的关于索引的部分代码也同时在m2eclipse中被使用，这些代码目前由Eugene Kuleshov开发。

16.2. 安装Nexus

16.2.1. 从Sonatype下载Nexus

你可以从http://nexus.sonatype.org找到关于Nexus的信息。要下载Nexus，访问http://nexus.sonatype.org/downloads/。点击下载链接，下载适用于你平台的存档文件。Nexus目前有ZIP和Gzip归档的TAR两种可用形式。

16.2.2. 安装Nexus

安装Nexus很简单，打开Nexus归档文件至一个目录。如果你正在本地工作站上安装Nexus，以测试它的运行，你可以将其安装至你的用户目录，或者随便什么你喜欢的地方；Nexus没有任何硬编码的目录，它能在任意目录运行。如果你下载了一个ZIP归档文件，运行：

```
$ unzip nexus-1.0.0-beta-3-bundle.zip
```

如果你下载了GZip归档的TAR文件，运行：

```
$ tar xvzf nexus-1.0.0-beta-3-bundle.tgz
```

如果你正在一个服务器上安装Nexus，你可能想要使用的目录不是你的用户目录。在Unix机器上，这可能是/usr/local/nexus-1.0.0-beta-3和一个指向Nexus目录的符号链接/usr/local/nexus。使用一个通用的符号链接来指向Nexus的某个特定版本是一个普遍的实践，它能让你更容易的将Nexus更新至最新的版本。

```
$ sudo cp nexus-1.0.0-beta-3-bundle.tgz /usr/local
$ cd /usr/local
$ sudo tar xvzf nexus-1.0.0-beta-3-bundle.tgz
$ ln -s nexus-1.0.0-beta-3 nexus
```

虽然对于Nexus的运行来说这不是必要的，你可能想要设置一个环境变量NEXUS_HOME，指向Nexus的安装目录。本章通过${NEXUS_HOME}的形式来引用这个位置。

16.2.3. 运行Nexus

当你启动Nexus，就是启动一个web服务器，它的默认地址是localhost:8081。Nexus在一个名为Jetty的servlet容器中运行，它使用一个名为Tanuki Java Service

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

Wrapper[1]的本地服务包裹器启动。这个服务包裹器可以被配置成以Windows服务或Unix守护线程的形式运行Nexus。要启动Nexus，你需要为你的平台找到合适的启动脚本。要查看可用平台的列表，查看${NEXUS_HOME}/bin/jsw目录的内容。

下面的例子展示了使用Mac OSX的脚本启动Nexus。首先我们列出${NEXUS_HOME}/bin/jsw的内容以查看可用的平台，然后我们用chmod命令使这个bin目录的内容可执行。Mac OSX包裹器通过调用app start启动，然后我们tail在${NEXUS_HOME}/container/logs中的wrapper.log。Nexus会初始化自己然后打印出一条信息说明它正在监听localhost:8081。

```
$ cd Nexus
$ ls ./bin/jsw/
aix-ppc-32/          linux-ppc-64/        solaris-sparc-32/
aix-ppc-64/          linux-x86-32/        solaris-sparc-64/
hpux-parisc-32/      linux-x86-64/        solaris-x86-32/
hpux-parisc-64/      macosx-universal-32/ windows-x86-32/
$ chmod -R a+x bin
$ ./container/bin/jsw/macosx-universal-32/app start
Nexus Repository Manager...
$ tail -f container/logs/wrapper.log
INFO  ... [ServletContainer:default] - Started SelectChannelConnector@0.0.0.0:8081
```

到目前为止，Nexus已经开始运行并监听端口8081。要使用Nexus，启动一个web浏览器然后输入URL：http://localhost:8081/nexus。点击web页面右上角的“Log In”链接，你应该看到如下的登陆对话框。

默认的NEXUS用户名和密码是"admin" 和 "admin123"。

[1] http://wrapper.tanukisoftware.org/doc/english/introduction.html

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

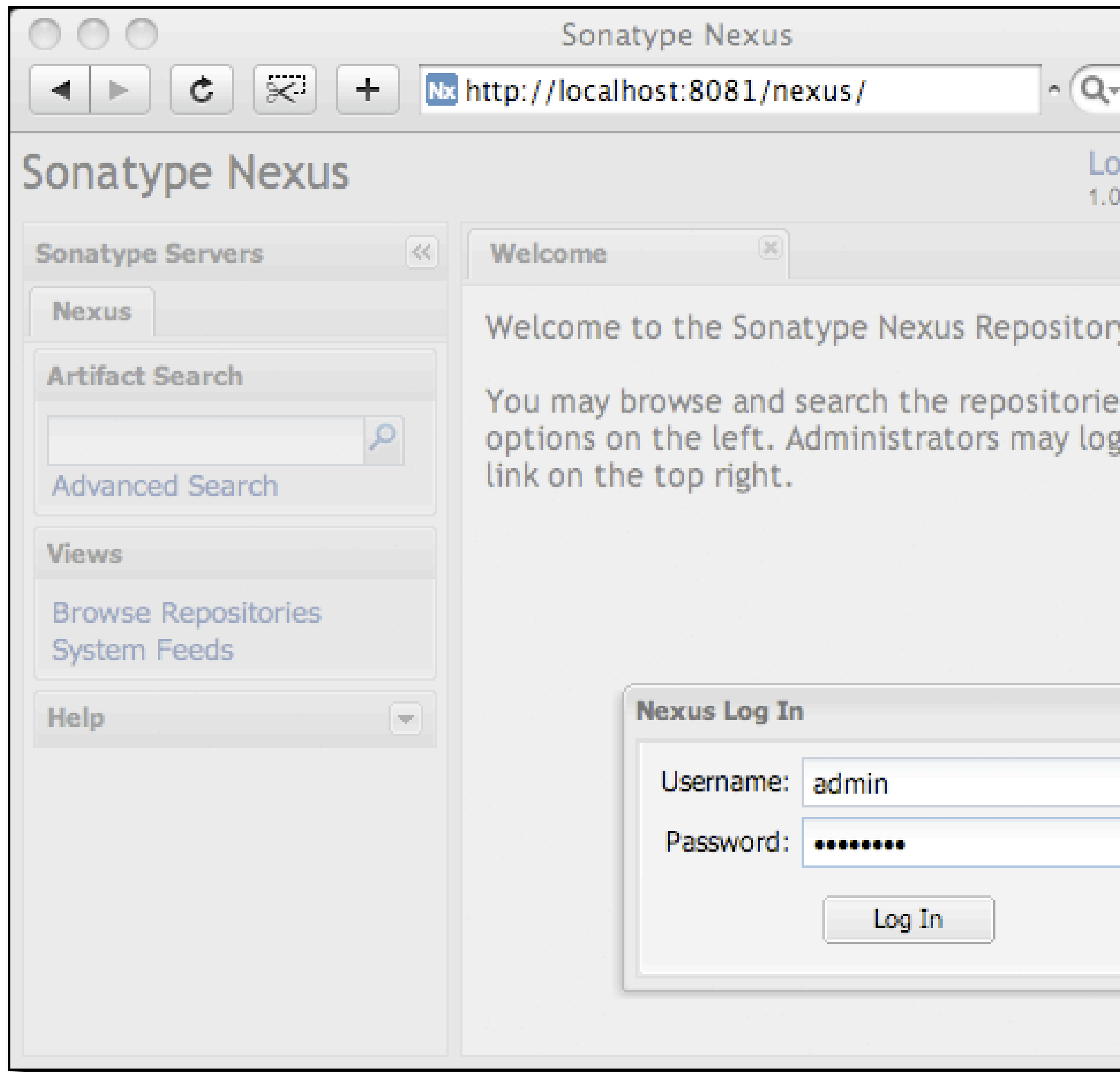


图 16.1. Nexus登陆窗口(默认 用户名/密码 是 admin/admin123)

16.2.4. 安装后检查单

Nexus带有默认的密码和仓库索引设置，它们都需要更改以满足你的安装需要（以及安全）。安装完并运行了Nexus后，你需要确认你完成了下列任务：

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

更改管理员密码和Email地址

默认的管理密码是admin123。对一个全新的Nexus安装，你首先要做的是更改这个密码。要更改默认的管理员登陆名"admin"及密码"admin123"，在浏览窗口的左边导航菜单中的Security部分点击Change Password。

配置SMTP设置

Nexus可以发送用来用户名和密码的email，要开启这个特性，你需要用SMTP主机和端口配置Nexus，以及相关的Nexus用来连接邮件服务器的认证参数。为此，载入如第 16.5. 节 “定制服务器配置”中的服务器配置对话框。

开启远程索引下载

Nexus带有三个重要的代理仓库，有中央Maven仓库，Apache快照仓库，和Codehaus快照仓库。它们中的每一个仓库都包含了数千（或数万）的构件，下载每个仓库的所有内容是不切实际的。处于这个原因，大部分仓库维护了一个编录了整个内容的Lucene索引，以提供快速和有效的搜索。Nexus使用这些远程索引搜索构件，但是默认设置我们关闭了索引下载。要下载远程索引，

1. 点击Administration菜单下面的Repositories，更改三个代理仓库的Download Remote Indexes为true。你需要为此载入如第 16.6 节 “维护仓库”中的对话框。

2. 在每个代理仓库上右击然后选择Re-index。这会触发Nexus下载远程的索引文件。

Nexus下载整个索引可能需要好几分钟，但是一旦你下载好之后，你就能够搜索Maven仓库的所有内容了。

注意

Sonatype想要确信没有创建一个会在默认情况下对中央仓库造成大量拥挤的产品。虽然大部分用户会想要开启远程索引下载，我们还是不想使之成为默认设置，当数百万用户下载一个新版本的Nexus继而不断的下载这个21MB的中央索引的时候，会制造对我们自己的拒绝服务攻击。如果你想要Nexus返回全部的搜索结果，你就必须显式的开启远程索引下载。

16.2.5. 为Redhat/Fedora/CentOS设置启动脚本

你可以将Nexus配置成自动启动，通过将app脚本拷贝到/etc/init.d目录。在一个Redhat变种的Linux系统上（Redhat, Fedora, 或者 CentOS），以root用户执行下列操作：

1. 复制${NEXUS_HOME}/bin/jsw/linux-ppc-64/app，或${NEXUS_HOME}/bin/jsw/linux-x86-32/app，或${NEXUS_HOME}/bin/jsw/linux-x86-64/app至/etc/init.d/nexus。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

2. 使/etc/init.d/nexus脚本可运行 —— chmod 755 /etc/init.d/nexus。

3. 编辑该脚本，更改下列变量。

   - 更改 APP_NAME 为 "nexus"
   - 更改 APP_LONG_NAME 为 "Sonatype Nexus"
   - 添加一个变量 NEXUS_HOME 指向你的 Nexus 安装目录
   - 添加一个变量 PLATFORM 内容包含 linux-x86-32， linux-x86-64， 或 linux-ppc-64
   - 更改 WRAPPER_CMD 为 ${NEXUS_HOME}/bin/jsw/${PLATFORM}/wrapper
   - 更改 WRAPPER_CONF 为 ${NEXUS_HOME}/conf/wrapper.conf
   - 更改 PIDDIR 为 /var/run.
   - 添加一个 JAVA_HOME 变量指向你的本地 Java 安装
   - 添加 /usr/lib/jvm/java-1.6.0-openjdk-1.6.0.0//bin 至 PATH

4. (可选) 设置 RUN_AS_USER 为 "nexus". 如果你这么做，你需要：

   - 创建一个 nexus 用户
   - 更改你的 nexus 安装目录的 Owner 和 Group 为 nexus

最后你应该有一个文件/etc/init.d/nexus，它拥有如下的一些列配置属性（假设你在/usr/local/nexus安装Nexus，你在/usr/java/latest安装了Java）：

```
JAVA_HOME=/usr/java/latest
PATH=/usr/local/bin:/usr/local/maven/bin:/usr/kerberos/sbin:/usr/kerberos/bin:/usr/
APP_NAME="nexus"
APP_LONG_NAME="Sonatype Nexus"
NEXUS_HOME=/usr/local/nexus
PLATFORM=linux-x86-64
WRAPPER_CMD="${NEXUS_HOME}/bin/jsw/${PLATFORM}/wrapper"
WRAPPER_CONF="${NEXUS_HOME}/conf/wrapper.conf"
PRIORITY=
PIDDIR="/var/run"
#RUN_AS_USER=nexus
```

这个脚本有一个适当的chkconfig指令，因此要添加Nexus为一个服务，你要做的是运行如下的命令：

```
$ cd /etc/init.d
$ chkconfig --add nexus
$ chkconfig --levels 345 nexus on
$ service nexus start
Starting Sonatype Nexus...
$ tail -f /usr/local/nexus/logs/wrapper.log
```

第二个命令添加nexus为一个服务，可以由service命令启动和停止，可以由chkconfig命令管理。chkconfig管理/etc/rc[0-6].d中的符号链接，当操作系统重启或者在运行级别中转换时，它们控制服务的启动和停止。第三个命令添加nexus至运行级别3，4，和5 。service命令启动Nexus，最后的命令追踪wrapper.log以验证Nexus成功启动。如果Nexus成功启动了你应该看到一个信息告诉你Nexus正在端口8001监听HTTP连接。

16.2.6. 升级Nexus版本

升级一个已安装的Nexus版本十分容易。每个Nexus版本有两个可用的归档文件可下载。完整的归档文件包含Nexus应用程序，Nexus启动脚本，以及用来保存仓库索引和远程仓库本地缓存的工作目录。如果你大量的使用Nexus，这个工作目录会包含数G的构件，你不会希望在每次升级Nexus的时候必须重新创建这个仓库。升级下载文件被创建成为用户提供一个方便的形式升级Nexus，它会保存Nexus数据；升级下载文件只包含Nexus应用程序代码。第一次你安装Nexus的时候，你下载完全的Nexus分发包，当你想要升级你的Nexus安装，同时保留你的仓库数据的时候，你只要下载升级分发包。

要升级Nexus，只要下载“upgrade”分发包，而非“bundle”分发包。升级分发包的内容存储在一个包含nexus版本号（如nexus-1.0.0-beta-3）的文件夹中。这个文件夹可以解开至$NEXUS_HOME/runtime/apps，不用覆盖当前安装版本的内容。

```
$ cd $NEXUS_HOME/runtime/apps
$ unzip nexus-1.0.0-beta-3-upgrade.zip
```

如果你下载了GZip归档的TAR文件，运行：

```
$ cd $NEXUS_HOME/runtime/apps
$ tar xvzf nexus-1.0.0-beta-3-upgrade.tgz
```

当升级归档文件解压至$NEXUS_HOME/runtime/apps后，你必须从之前的Nexus版本复制配置文件至新安装的版本。从$NEXUS_HOME/runtime/apps/nexus/conf/nexus.xml复制nexus.xml至$NEXUS_HOME/runtime/apps/nexus-1.0.0-beta-3/conf。你应该也复制所有你自定义的日志配置文件log4j.properties和jul-logging.properties。在你从当前的Nexus版本复制了配置文件至新版本的Nexus后，停止Nexus服务器。

现在，你需要重命名$NEXUS_HOME/runtime/apps/nexus目录为一个反映它旧版本号的名称。比如，在这个例子中$NEXUS_HOME/runtime/apps/nexus将成为$NEXUS_HOME/

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

runtime/apps/nexus-1.0.0-beta-3。然后，将你新版本改为$NEXUS_HOME/runtime/apps/nexus。在Unix系统上，你需要创建一个符号链接$NEXUS_HOME/runtime/apps/nexus指向$NEXUS_HOME/runtime/apps/nexus-1.0.0-beta-2。在Windows系统上，你可能需要复制$NEXUS_HOME/runtime/apps/nexus-1.0.0-beta-2至$NEXUS_HOME/runtime/apps/nexus。在你用新版本的Nexus交换了旧版本的Nexus后，你应该能使用启动脚本启动Nexus。Nexus启动之后，检查$NEXUS_HOME/logs/wrapper.log。Nexus初始化之后，它会打印出Nexus版本号。

16.3. 使用Nexus

Nexus为那些只需要搜索，浏览构件，以及查阅系统RSS源的用户提供了匿名访问。匿名访问级别更改了导航菜单，以及当你在一个仓库上右击时可用的选项。这种只读访问显示了如图 16.2 “匿名用户的Nexus界面”的用户界面。

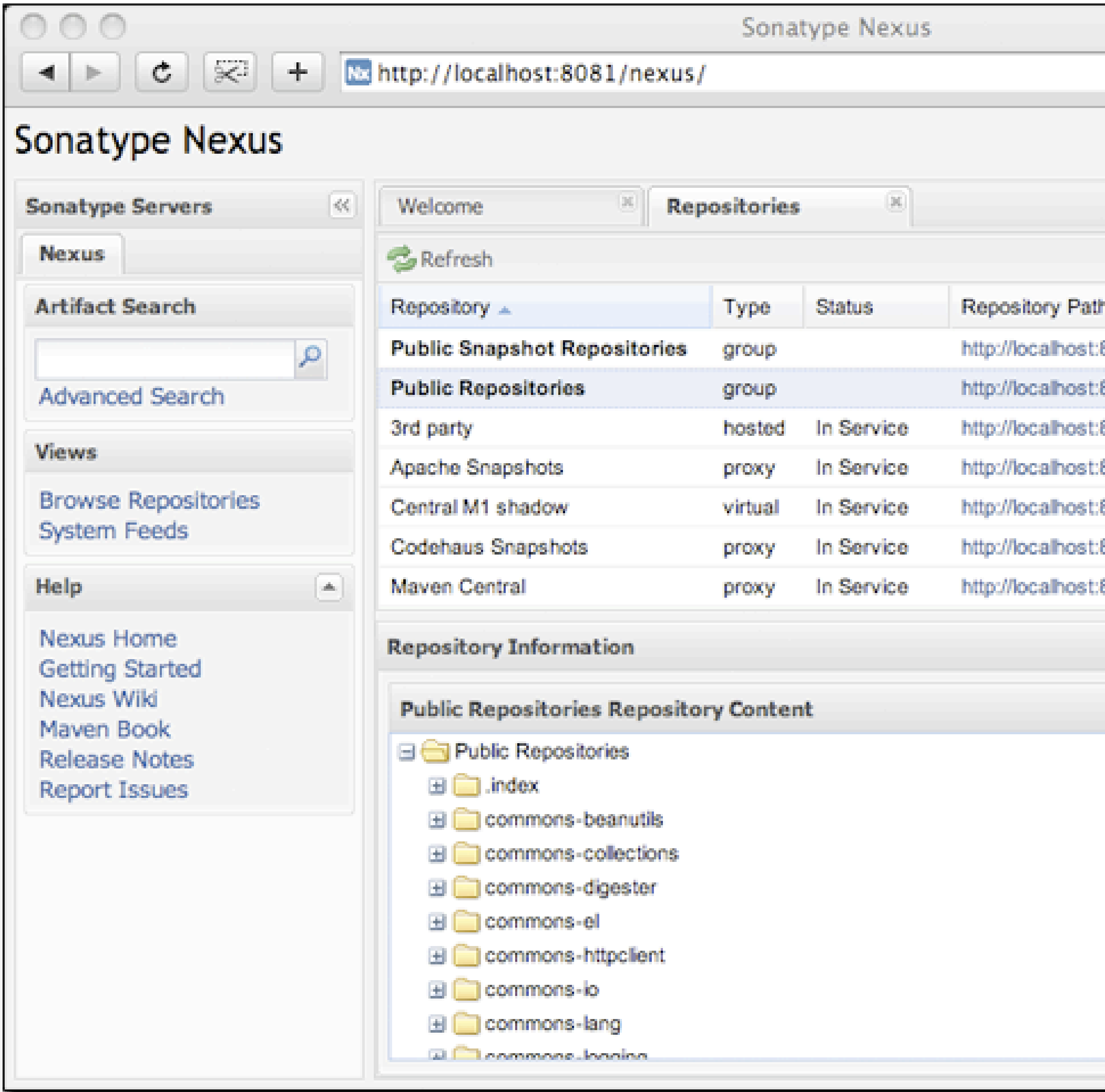


图　16.2.　匿名用户的Nexus界面

16.3.1.　浏览仓库

Nexus最直接的用途之一就是浏览Maven仓库的结构。如果你点击Views菜单下的Browse Repositories菜单项。图　16.3 “浏览一个Nexus仓库”中的上面一半给你显示了带有仓库类型和仓库状态的组列表和仓库列表。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

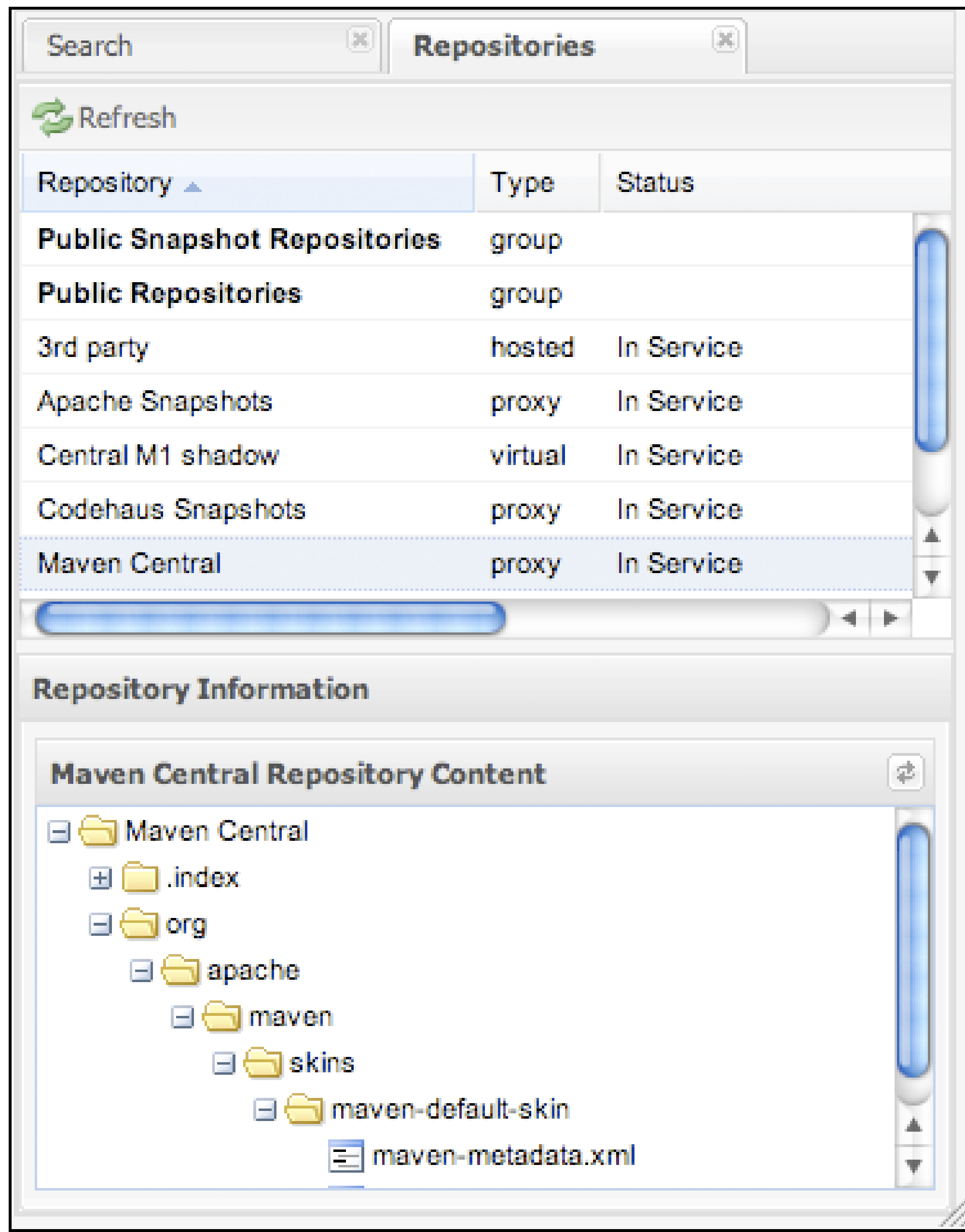


图 16.3. 浏览一个Nexus仓库

当你浏览一个仓库的时候，你可以在任意一个文件上右击然后直接下载到你本地。这能让你手工获取某个特定的构件，或者在浏览器中检查一个POM文件。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

16.3.2. 浏览组

Nexus包含排序好的仓库组，它们能让你通过一个单独的URL来暴露一系列的仓库。通常情况下，一个组织会指向两个默认的Nexus组：Public Repositories组和Public Snapshot Repositories组。很多最终用户不需要知道哪些构件来自哪个特定的仓库，他们只需要能够浏览公共仓库组就可以了。为了支持这个用例，Maven允许你浏览一个Nexus组的内容，它就像是一归并而来的树状的仓库。图 16.4 “浏览一个Nexus组”显示了这个浏览界面，其中一个Nexus组被选中以浏览。对用户体验来说，浏览一个Nexus组和浏览一个Nexus仓库没任何区别。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

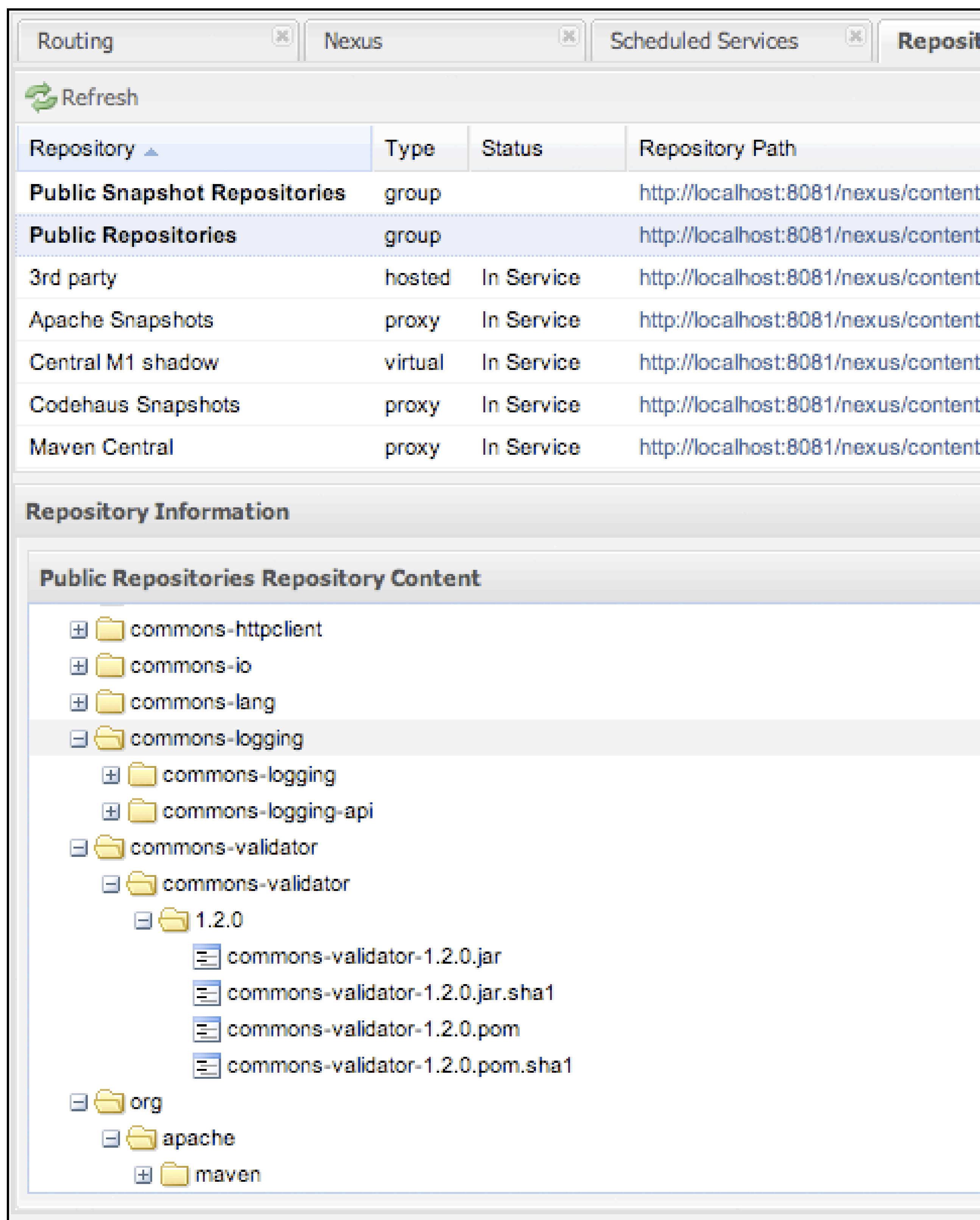


图 16.4. 浏览一个Nexus组

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

16.3.3. 搜索构件

在左边的导航区域，紧靠放大镜有一个构件搜索输入框。要通过groupId或者artifactId搜索一个构件，输入一些文本然后点击放大镜。输入字段"maven"然后点击放大镜会产生如图 16.5 “关键词为"maven"的构件搜索结果”的搜索结果。

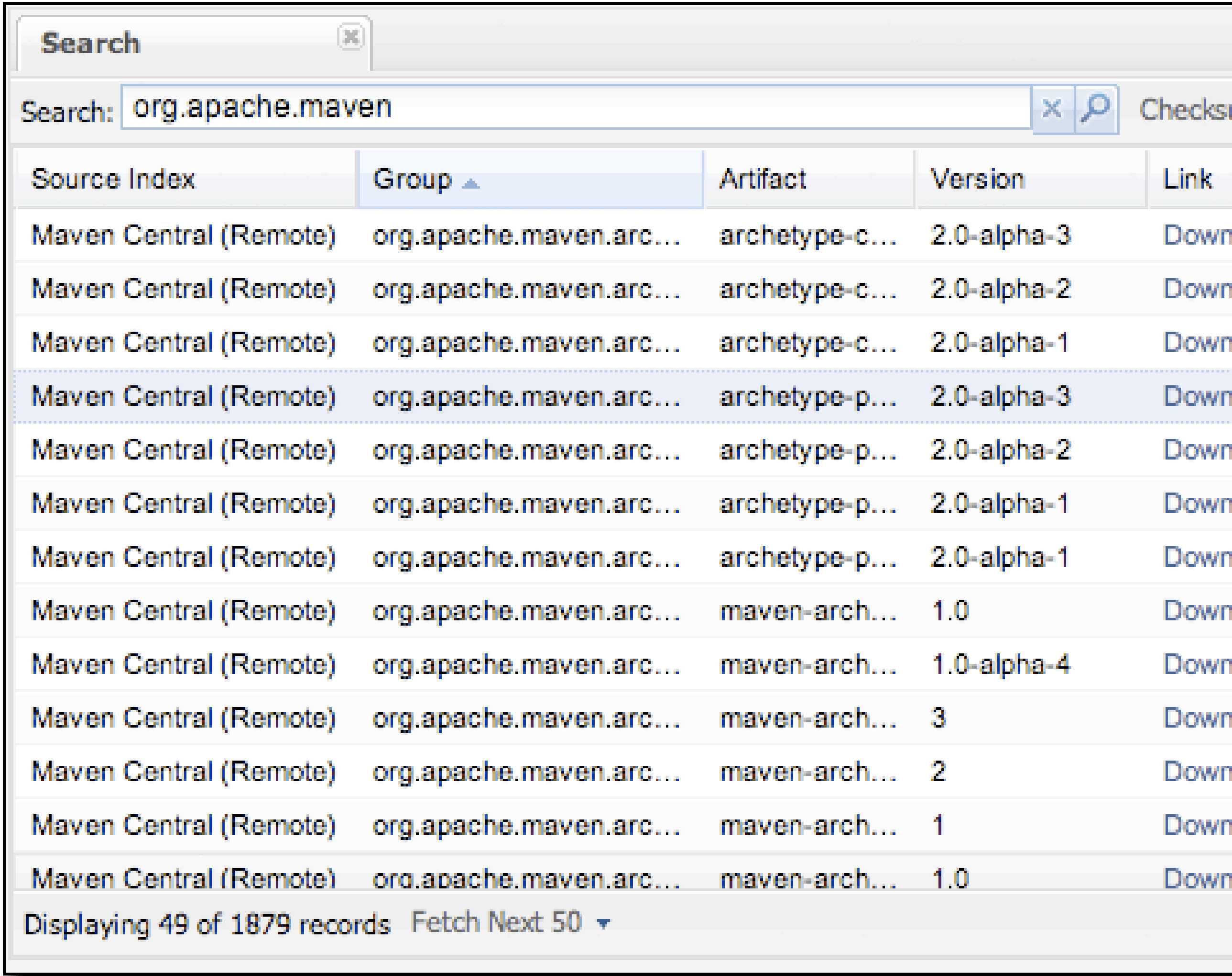


图 16.5. 关键词为"maven"的构件搜索结果

在你找出你在要找的构件之后，你可以点击Download链接来下载这个构件。Nexus每次为你显示50条结果，并且为你浏览其它搜索结果在底部提供了链接。如果你更喜欢看到所有匹配构件的列表，你可以在搜索结果面板底部的下拉菜单中选择Fetch All。

除了通过一个groupId或者一个artifactId搜索，Nexus还有一个功能能让你通过校验和来搜索一个构件。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

警告

让我来猜一下？你安装了Nexus，使用了搜索框，输入了一个构件的group的名字，按下搜索，然后什么都没看见。没有结果。Nexus默认不会去获取远程仓库索引，你需要为那三个Nexus自带的仓库激活远程索引的下载。没有这些索引，没有没东西可以搜索。你可以在第 16.2.4 节 “安装后检查单”中查找激活索引下载的指令。

16.3.4. 浏览系统RSS源

Nexus提供了一些捕捉系统事件的RSS源，你可以通过点击View菜单下的System Feeds来浏览它们。如图 16.6 “浏览Nexus系统信息源”中的面板。你可以使用这些简单的界面来浏览最近Nexus中发生的关于构件部署，构件缓存，存储变化的报告。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

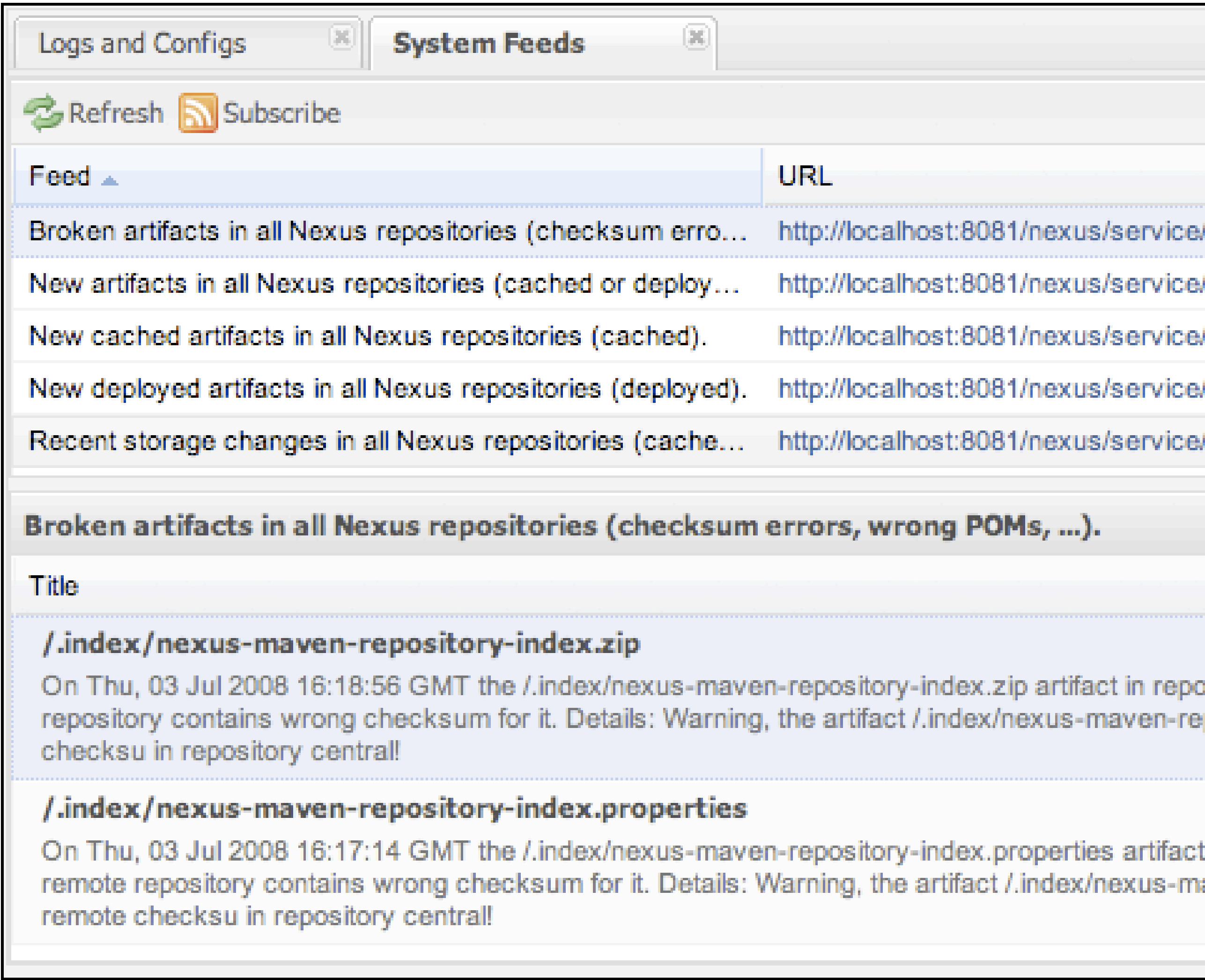


图 16.6. 浏览Nexus系统信息源

如果你正在一个很大的组织工作，很多开发团队往同样一个Nexus实例部署构件，这些信息源就非常有用。有了这样的准备，所有组织开发人员可以为新部署的构件订阅RSS信息源，以确保当一个新的发布版提交到Nexus后所有的人都知道。将系统事件暴露成RSS信息源也将大门向其他人开启，包括一些对该信息更富创意的使用，如将Nexus与外部的自动测试系统想连。要访问某个特定信息源的RSS，在System Feeds观察面板中选择一个信息源然后点击Subscribe按钮。Nexus会在你浏览器中载入这个RSS信息源，然后你可以在你最喜欢的RSS阅读器中订阅这个信息源。

在系统信息源视图中有6个可用的信息源，每一个信息源都有一个类似于下面的URL：

```
http://localhost:8081/nexus/service/local/feeds/recentChanges
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

其中recentChanges将会被你试图阅读的信息源标识所替换。可能的系统信息源包括：

表 16.1. 可用的系统信息源

信息源标识符	描述
brokenArtifacts	校验和不匹配，找不到校验和，不可用的POM
recentCacheOrDeployments	所有仓库中有新的构件（从远程缓存的或者部署上去的）
recentlyCached	所有仓库中有新的缓存构件
recentlyDeployed	所有仓库中有新的部署的构件
recentChanges	所有缓存，部署，或者删除动作
systemRepositoryStatusChanges	自动或者用户发起的变更（服务失效和阻塞的代理）
systemChanges	启动Nexus，更改配置，重新编制索引，以及属性重建

16.3.5. 浏览日志文件和配置

日志和配置文件只有在管理员用户的Views菜单中可见。点击该选项会看到如图 16.7 “浏览Nexus日志和配置文件”中的对话框。在这个屏幕你可以通过点击Download按钮旁边的下拉选择菜单来查看一下的日志和配置文件。

nexus.log
: 把它想成是一个Nexus的总体的应用程序日志。除非你是管理员用户，否则你可能不会对这个日志的信息有什么兴趣。如果你正试图调试一个错误，或者你有Nexus中未发现的bug。你会使用这个日志查看器来诊断Nexus的问题。

nexus-rest-0.log
: 核心的Nexus服务实际上是一堆REST服务，你正使用的UI只是和这些REST服务交互以配置和查看Nexus的组及仓库。这个日志文件反映了由Nexus UI和Nexus REST服务交互所生成的活动。

nexus.xml
: 这个XML文件包含了大部分你所使用的Nexus实例的配置数据。它被存储在${NEXUS_HOME}/runtime/apps/nexus/conf/nexus.xml。

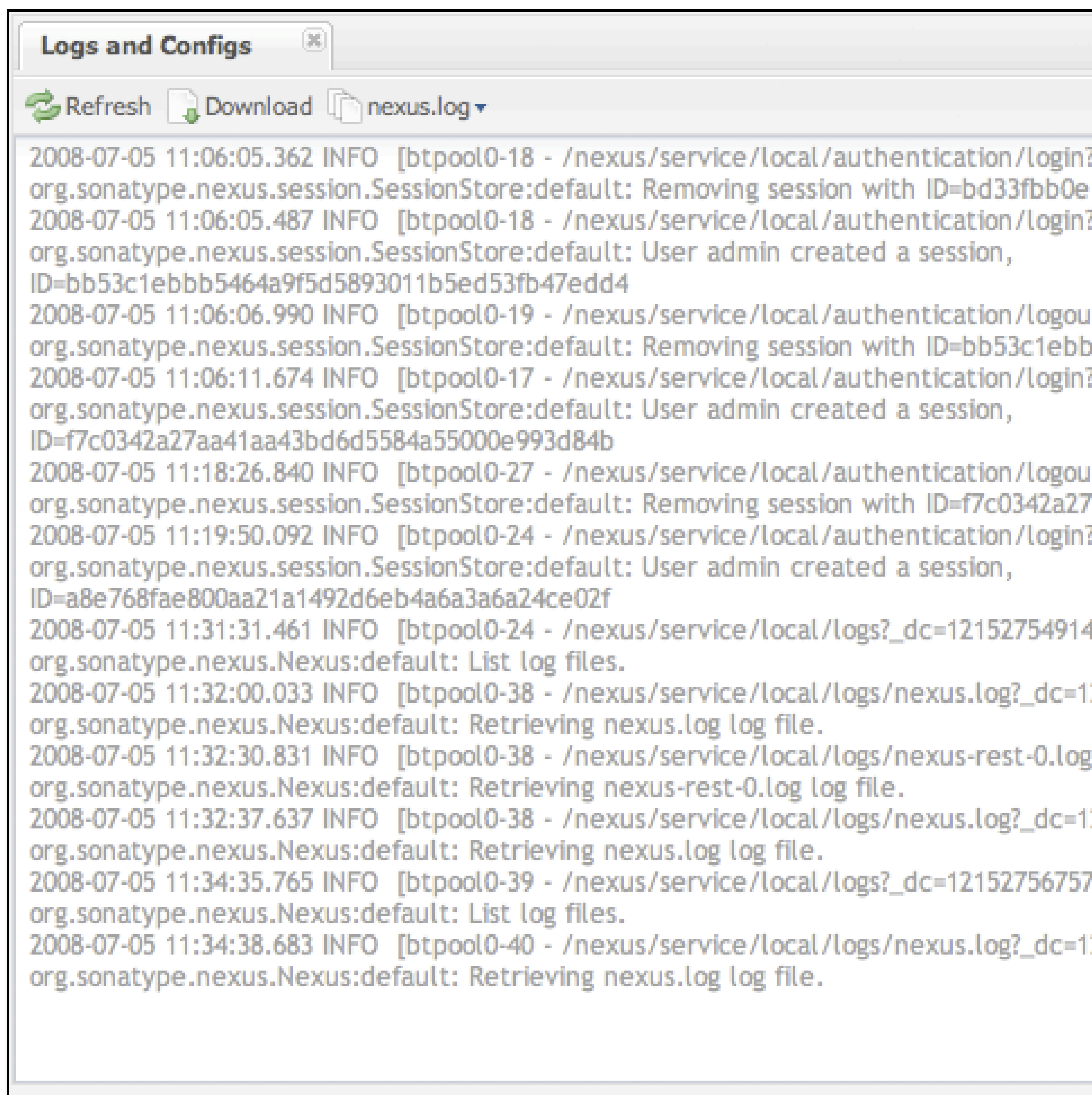


图 16.7. 浏览Nexus日志和配置文件

16.3.6. 更改你的密码

如果你拥有适当的安全权限，你还会在浏览器的左边看到一个可以更改你密码的选项。要更改你的密码，点击change password，提供你现在的密码，然后输入一个新密码。当你点击Change Password后，你的Nexus密码就被更改了。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

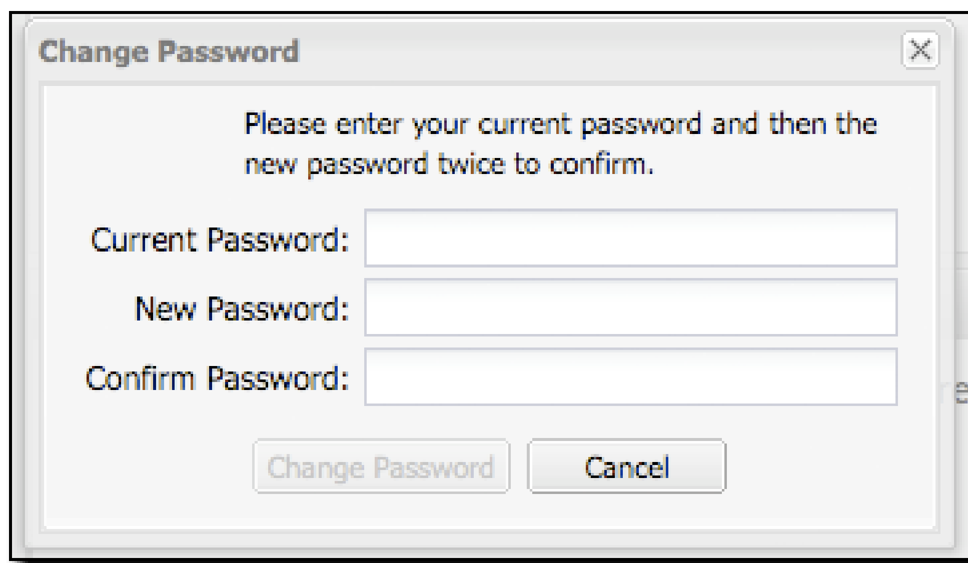


图 16.8. 更改你的Nexus密码

16.4. 配置Maven使用Nexus

要使用Nexus，你需要配置Maven去检查Nexus而非公共的仓库。为此，你需要编辑在你的`~/.m2/settings.xml`文件中的mirror配置。首先，我们会演示如何配置Maven去检查你的Nexus安装而非直接从中央Maven仓库获取构件。在我们覆盖了中央仓库并演示了Nexus可以工作之后，我们会转回来，提供一个更实际的，包含发布版和快照版的配置集合。

16.4.1. 使用Nexus中央代理仓库

要配置Maven去查阅Nexus而非中央Maven仓库，在你的`~/.m2/settings.xml`文件中添加如例 16.1 "为Nexus配置Maven Settings (~/.m2/settings.xml)"的mirror配置。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

例 16.1. 为Nexus配置Maven Settings (~/.m2/settings.xml)

```
<?xml version="1.0"?>
<settings>
  ...
  <mirrors>
    <mirror>
      <id>Nexus</id>
      <name>Nexus Public Mirror</name>
      <url>http://localhost:8081/nexus/content/groups/public</url>
      <mirrorOf>central</mirrorOf>
    </mirror>
  </mirrors>
  ...
</settings>
```

在你将Nexus配置成所有仓库的镜像之后，Maven现在会从本地的Nexus安装查阅，而非去外面查阅中央Maven仓库。如果对Nexus有一个构件请求，本地的Nexus安装会提供这个构件。如果Nexus没有这个构件，Nexus会从远程仓库获取这个构件，然后添加至远程仓库的本地镜像。

要测试Nexus如何工作的，从你的本地Maven仓库中删除一个目录，然后运行Maven构建。如果你删除了~/.m2/repository/org，你会删除一大堆的依赖（包括Maven 插件）。下次你运行Maven的时候，你应该看到如下的信息：

```
$ mvn clean install
...
Downloading: http://localhost:8081/nexus/content/groups/public/...
3K downloaded
```

这个输出应该能让你相信Maven正和你本地的Nexus通讯，而非向外面的中央Maven仓库获取构件。在你基于本地的Nexus运行过一些构建之后，你就可以浏览缓存在你本地Nexus中的内容。登陆Nexus然后点击导航菜单的左边的构件搜索。在搜索框中输入"maven"，你应该能看到一些像下面的内容。

16.4.2. 使用Nexus作为快照仓库

第 16.4.1 节 “使用Nexus中央代理仓库”中的Maven配置能让你使用Nexus公共组，这个组从4个由Nexus管理的仓库解析构件，但是它不让你查阅public-snapshots组，该组包括了Apache和Codehaus的快照版。要配置Maven让它为发布版和插件都使用Nexus，你必须配置Maven，通过往你的Maven文件~/.m2/settings.xml中添加如下的镜像配置，使其查阅Nexus的组。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

```
<settings>
    <mirror>
      <!--This is used to direct the public snapshots repo in the
          profile below over to a different nexus group -->
      <id>nexus-public-snapshots</id>
      <mirrorOf>public-snapshots</mirrorOf>
      <url>http://localhost:8081/nexus/content/groups/public-snapshots</url>
    </mirror>
    <mirror>
      <!--This sends everything else to /public -->
      <id>nexus</id>
      <mirrorOf>*</mirrorOf>
      <url>http://localhost:8081/nexus/content/groups/public</url>
    </mirror>
  </mirrors>
  <profiles>
    <profile>
      <id>development</id>
      <repositories>
        <repository>
          <id>central</id>
          <url>http://central</url>
          <releases><enabled>true</enabled></releases>
          <snapshots><enabled>true</enabled></snapshots>
        </repository>
      </repositories>
     <pluginRepositories>
        <pluginRepository>
          <id>central</id>
          <url>http://central</url>
          <releases><enabled>true</enabled></releases>
          <snapshots><enabled>true</enabled></snapshots>
        </pluginRepository>
      </pluginRepositories>
    </profile>
    <profile>
      <!--this profile will allow snapshots to be searched when activated-->
      <id>public-snapshots</id>
      <repositories>
        <repository>
          <id>public-snapshots</id>
          <url>http://public-snapshots</url>
          <releases><enabled>false</enabled></releases>
          <snapshots><enabled>true</enabled></snapshots>
        </repository>
      </repositories>
     <pluginRepositories>
        <pluginRepository>
```

```
$ mvn -Ppublic-snapshots clean install
```

```
          <url>http://public-snapshots</url>
          <releases><enabled>false</enabled></releases>
          <snapshots><enabled>true</enabled></snapshots>
        </pluginRepository>
      </pluginRepositories>
    </profile>
  </profiles>
  <activeProfiles>
```

在例 "16.2 "配置Maven使其为发布版和快照版使用Nexus"中我们定义了两个

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

16.4.3. 为缺少的依赖添加仓库

如果你已经将你的Maven settings.xml配置成使用Nexus作为所有公共仓库和所有公共快照仓库的镜像，你可能会遇到一些项目不能够从你的本地Nexus获取需要的构件。这很常见，因为你经常会构建一些在pom.xml中自定义一组repositories和snapshotRepositories的项目。如果你正在构建开源项目，或者往你的配置中添加了自定义的第三方Maven仓库，那么这种情况就会发生。

作为一个例子，让我们试试从我们签出的源代码构件Apache Shindig。什么是Apache Shindig?对该例来说这不重要；我们需要的是一个能很容易签出和构建的样例项目。如果你实在很想知道，Shindig是在Apache Incubator中的一个围绕Google的OpenSocial API的项目。Shindig目标是提供一个允许人们运行OpenSocial小工具的容器。它给我们提供了一个有趣的样例工程，因为它有一些没有被加入到中央Maven仓库的组件，于是依赖于一些自定义的Maven仓库，使用Shindig我们可以向你展示当Nexus没有你要的构件的时候会发生什么，以及你能够使用怎样的步骤来给Nexus添加仓库。

下面的样例假设你已经安装了Subversion，并且你正在命令行运行Subversion。我们使用Subversion从Apache Incubator签出Apache Shindig然后尝试构建它。为此，执行下面的命令：

```
$ svn co http://svn.apache.org/repos/asf/incubator/shindig/trunk shindig
... Subversion will checkout the trunk of Apache Shindig ...
$ cd shindig
$ mvn install
... Maven will build Shindig ...
Downloading: http://localhost:8081/nexus/content/groups/public/caja/caja/r820/caja-
Downloading: http://localhost:8081/nexus/content/groups/public/caja/caja/r820/caja-
[INFO] ------------------------------------------------------------------------
[ERROR] BUILD ERROR
[INFO] ------------------------------------------------------------------------
[INFO] Failed to resolve artifact.

Missing:
----------
1) caja:caja:jar:r820

  Try downloading the file manually from the project website.

...
----------
1 required artifact is missing.

for artifact:
  org.apache.shindig:gadgets:war:1-SNAPSHOT
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛


```
from the specified remote repositories:
  oauth (http://oauth.googlecode.com/svn/code/maven),
  central (http://central),
  apache.snapshots (http://people.apache.org/repo/m2-snapshot-repository),
  caja (http://google-caja.googlecode.com/svn/maven)
```

这个构建失败了因为它下载不到一个构件。这个构件有一个group标识符为caja，artifactId是caja，版本是r820。这是一个存在于自定义仓库http://google-caja.googlecode.com/svn/maven中的一个构件。Maven没能够下载到这个构件是因为你的settings.xml被配置成指引所有的镜像至位于我们Nexus安装的public和public-snapshots组。即使Apache Shindig的pom.xml定义了一个仓库并且将其指向了http://google-caja.googlecode.com/svn/maven，Nexus不会从一个它不知道的仓库中去获取构件。事实上，关于这次构建有两个仓库Nexus不知道：caja和oauth。Caja[2]和OAuth[3]是两个仍然处于开发中的类库。两个项目都被“发布”了，而且Shindig所依赖的版本当然不是快照版，但是这些项目没有被发布到中央Maven仓库。在我们能构建这个项目之前，我们需要想办法让Nexus知道这些仓库。

有两种方法可以解决这个问题。首先，你可以更改你的以settings.xml覆盖特定的仓库定义符。你可以更改settings.xml中的mirrorOf元素为"central"，而非让Nexus public组 mirrorOf所有的仓库。如果你这么做了，Maven就会试图直接从oauth和caja仓库下载依赖。这行得通，因为Maven只会为那些匹配settings.xml中mirrorOf元素的仓库去查阅Nexus。如果Maven看到一个仓库定义符caja或者oauth，而且没有在你的settings.xml中看到一个镜像，它会直接去连接这个仓库。

第二种方法，更有趣的选择是添加这些仓库至Nexus，并且添加这些仓库至public组。

16.4.4. 添加一个新的仓库

要添加caja仓库，以管理员登陆Nexus，在左边导航菜单Configuration部分中点击Repositories链接。点击这个链接后会看到一个窗口列出了所有Nexus所知道的仓库。之后你想要创建一个新的代理仓库。为此，点击在仓库列表正上方的Add链接。点击单词Add右边的朝下的箭头，会看到一个下拉菜单，带有选项：Hosted，Proxy，和Vitual。既然你要创建一个代理仓库，点击Proxy。之后，你会看到一个如图 16.9 “添加一个Nexus仓库”的页面。填充那些必填字段，Repository ID为"caja"，Repository Name为"Google Caja"。设置Repository Policy为"Release"，以及Remote Storage Location为http://google-caja.googlecode.com/svn/maven。

[2] http://code.google.com/p/google-caja/
[3] http://code.google.com/p/oauth/

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

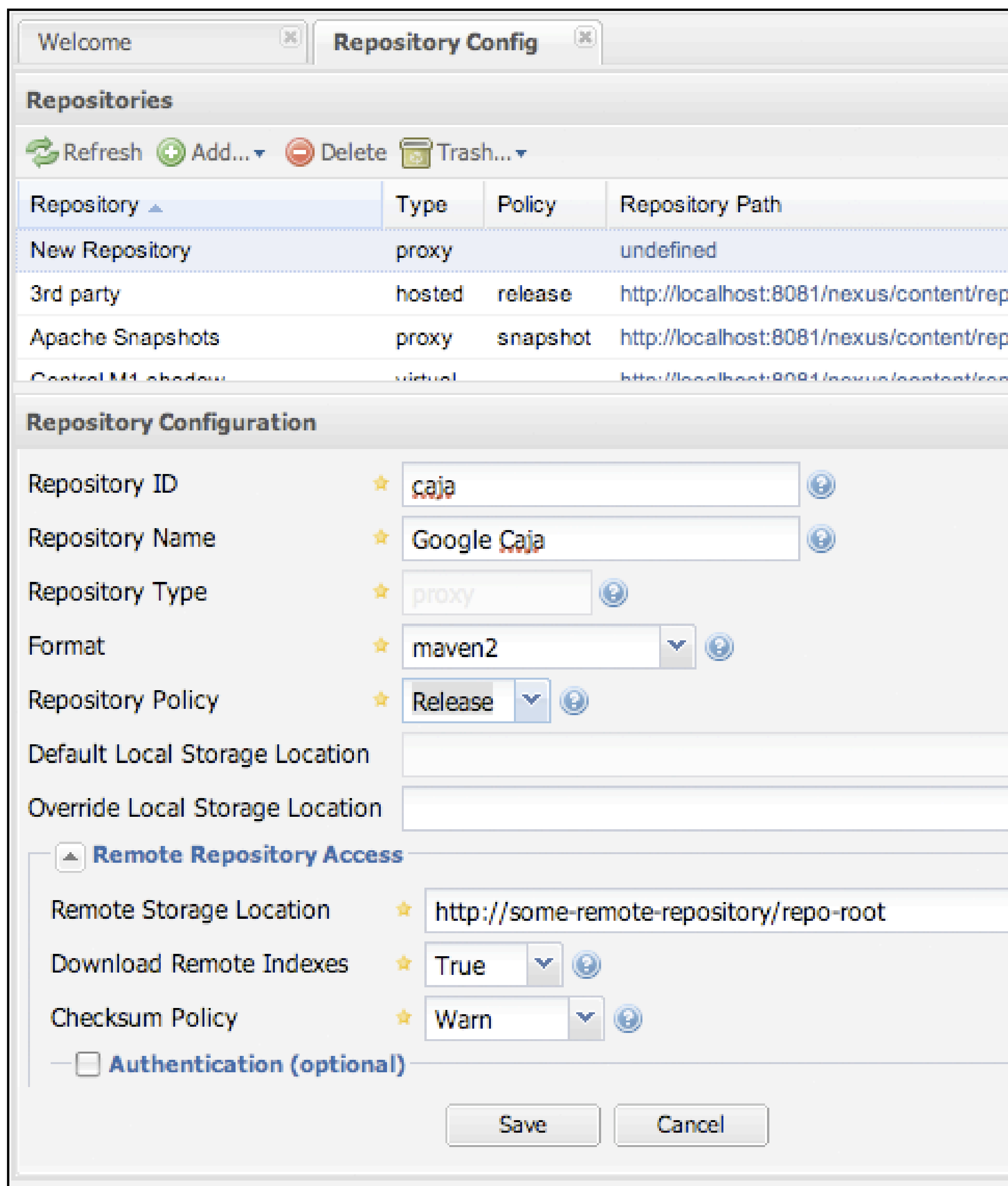


图 16.9. 添加一个Nexus仓库

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

在你填写完这个页面之后，点击Save按钮。Nexus就会接受这个caja代理仓库的配置。为oauth仓库重复同样的工作。创建一个仓库，Repository ID为oauth，选择Release policy，Remote Storage Location为http://oauth.googlecode.com/svn/code/maven。

16.4.5. 添加一个仓库至一个组

下一步你需要做的是添加这些新的仓库至public Nexus组。为此，点击左边导航菜单中Configuration部分的Groups链接。当你看到组管理页面后，点击public repositories组，你应该能看到如图 16.10 “添加新的仓库至一个Nexus组”的页面。

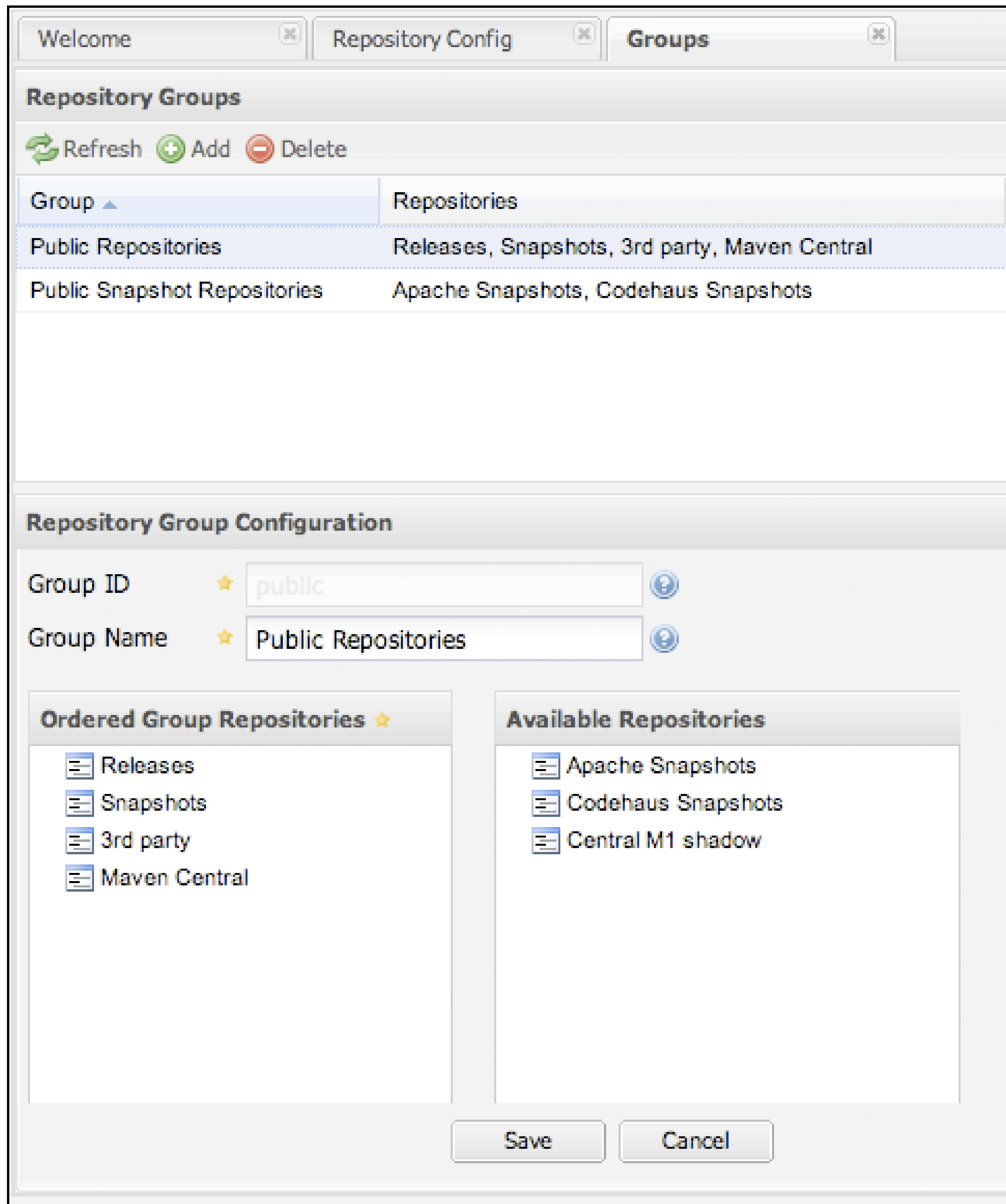


图 16.10. 添加新的仓库至一个Nexus组

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

Nexus使用了一个十分有趣的，名为ExtJS[4]的Javascript小工具类库。ExtJS提供了许多有趣的UI小工具，能为用户提供丰富的交互体验。要添加这两个新的仓库至public Nexus组，在可用仓库列表中找到仓库，点击你想要添加的仓库然后拖拉进Ordered Group Repositories。一旦仓库在Ordered Group Repositories列表中，你可以点击并拖拉列表中的仓库，以改变为匹配构件进行搜索的仓库的顺序。在Google Caja和Google OAuth项目仓库被添加到public Nexus组之后，你应该能够构建Apache Shindig并观察到Maven从各自的仓库下载Caja和OAuth。

16.5. 配置Nexus

本节展示的很多配置页面只对管理员可用。Nexus允许管理员用户自定义仓库列表，创建仓库组，自定义服务器设置，以及创建Maven用来包含或排除某个仓库构件的路线或者“规则”。

16.5.. 定制服务器配置

在一个实际的Nexus安装中，你可能会想要自定义管理员密码，而非使用“admin123”，你可能会想要复写Nexus用来存储仓库数据的默认目录。为此，以管理员用户登陆然后点击左边导航菜单Administration下面的Server。服务器配置界面如图 16.11 “Nexus服务器配置”显示。

[4] http://extjs.com/

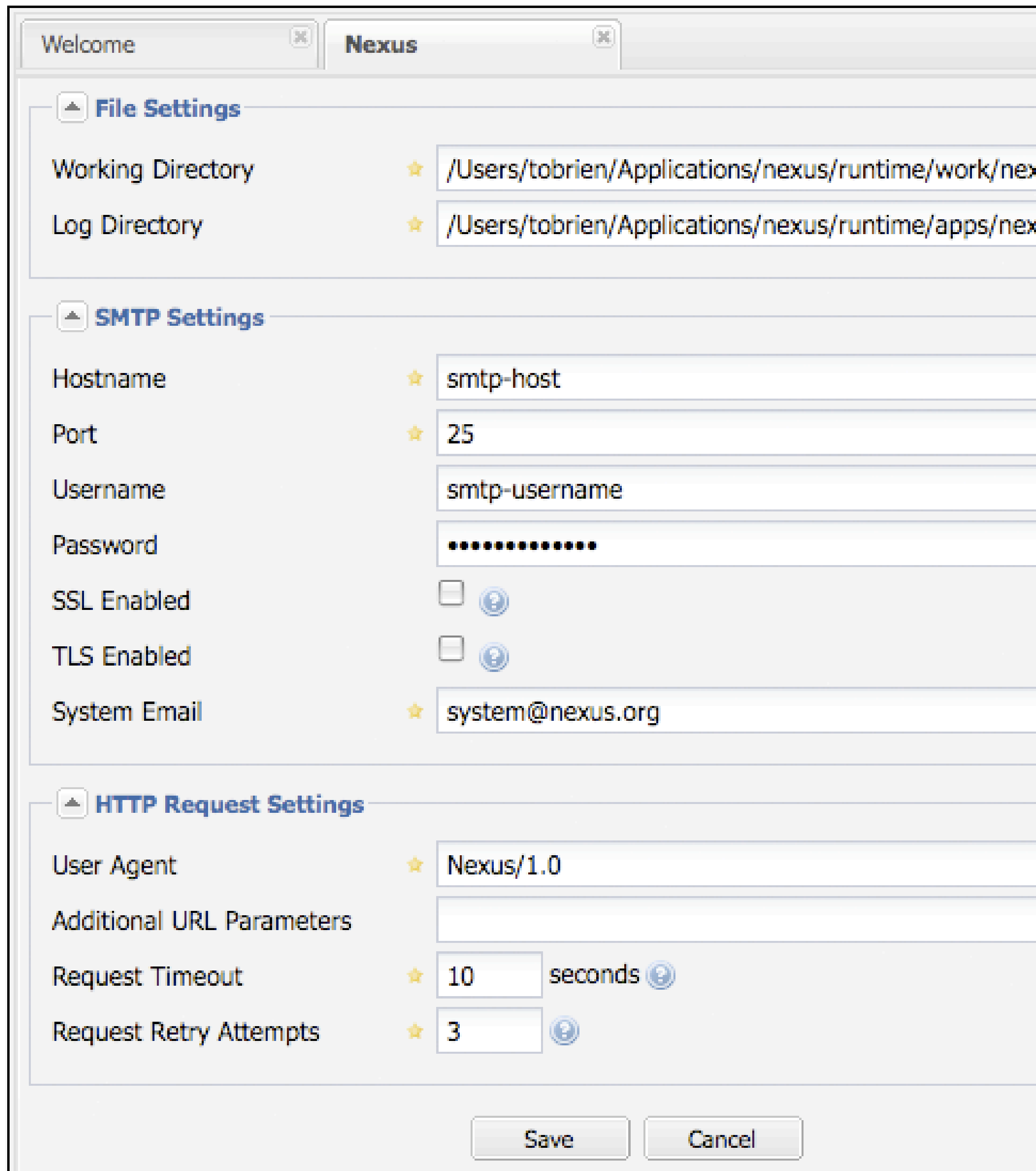


图 16.11. Nexus服务器配置

该页面能让你更改：

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

管理员密码
: 默认的管理员密码是admin123。如果你填写这个字段并点击了Save按钮，你及更改了这个Nexus安装的管理员密码。

工作目录
: 在File Settings组下面，你可以自定义工作目录。如果你的Nexus安装将要作为很大的仓库的镜像，而且你想要将你的工作目录放到另外一个硬盘分区，你可能会想要自定义工作目录。

日志目录
: 你可以改变Nexus寻找日志的位置。在一个Unix机器上，一个通常的实践是将日志文件放到/var/log。如果你遵循这个实践，你可以使用适当的权限来创建一个/var/log/nexus目录。注意这个设置并不会更改Nexus记日志的目录，它仅仅是告诉Nexus去哪里寻找日志。要更改写日志的位置，你需要修改在你Nexus安装的runtime/apps/nexus/conf目录下jul-logging.properties的和log4j.properties文件。

User Agent
: 这是Nexus用来生成HTTP请求的标识符。如果Nexus需要用一个HTTP代理，而且这个代理只有当User Agent设置成某个特定值才能工作，你就要更改这个设置。

额外的URL参数
: 这是一列放在对远程仓库的GET请求后面的附加参数。你可以用它来添加对请求的定义信息。

请求超时
: 这是当Nexus和外部，远程的仓库交互时等待一个请求成功的时间。

请求重试次数
: 当遇到一个失败的HTTP请求时，Nexus会重试的次数。

代理主机和代理端口
: 如果你的组织需要使用一个HTTP代理服务器，你可以在这里提供代理主机和代理端口。

代理认证
: 这一部分配置能让你提供代理认证信息，如用户名和密码，或者用来访问HTTP代理的密钥。

16.5.2. 管理仓库

要管理Nexus提供的仓库，以管理员用户登陆然后点击左边导航菜单Administration下面的Repositories。Nexus提供了三种不同的仓库。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

代理仓库

一个代理仓库是对远程仓库的一个代理。默认情况下，Nexus自带了如下配置的代理仓库：

Apache Snapshots

这个仓库包含了来自于Apache软件基金会的快照版本。http://people.apache.org/repo/m2-snapshot-repository

Codehaus Snapshots

这个仓库包含了来自于Codehaus的快照版本。 http://snapshots.repository.codehaus.org/

Central Maven Repository

这是中央Maven仓库（发布版本）。 http://repo1.maven.org/maven2/

宿主仓库

一个宿主仓库是由Nexus托管的仓库。Maven自带了如下配置的宿主仓库。

3rd Party

这个宿主仓库应该用来存储在公共Maven仓库中找不到的第三方依赖。这种依赖的样例有：你组织使用的，商业的，私有的类库如Oracle JDBC驱动。

Releases

这个宿主仓库是你组织公布内部发布版本的地方。

Snapshots

这个宿主仓库是你组织发布内部快照版本的地方。

虚拟仓库

一个虚拟仓库作为Maven 1的适配器存在。Nexus自带了一个central-m1虚拟仓库。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

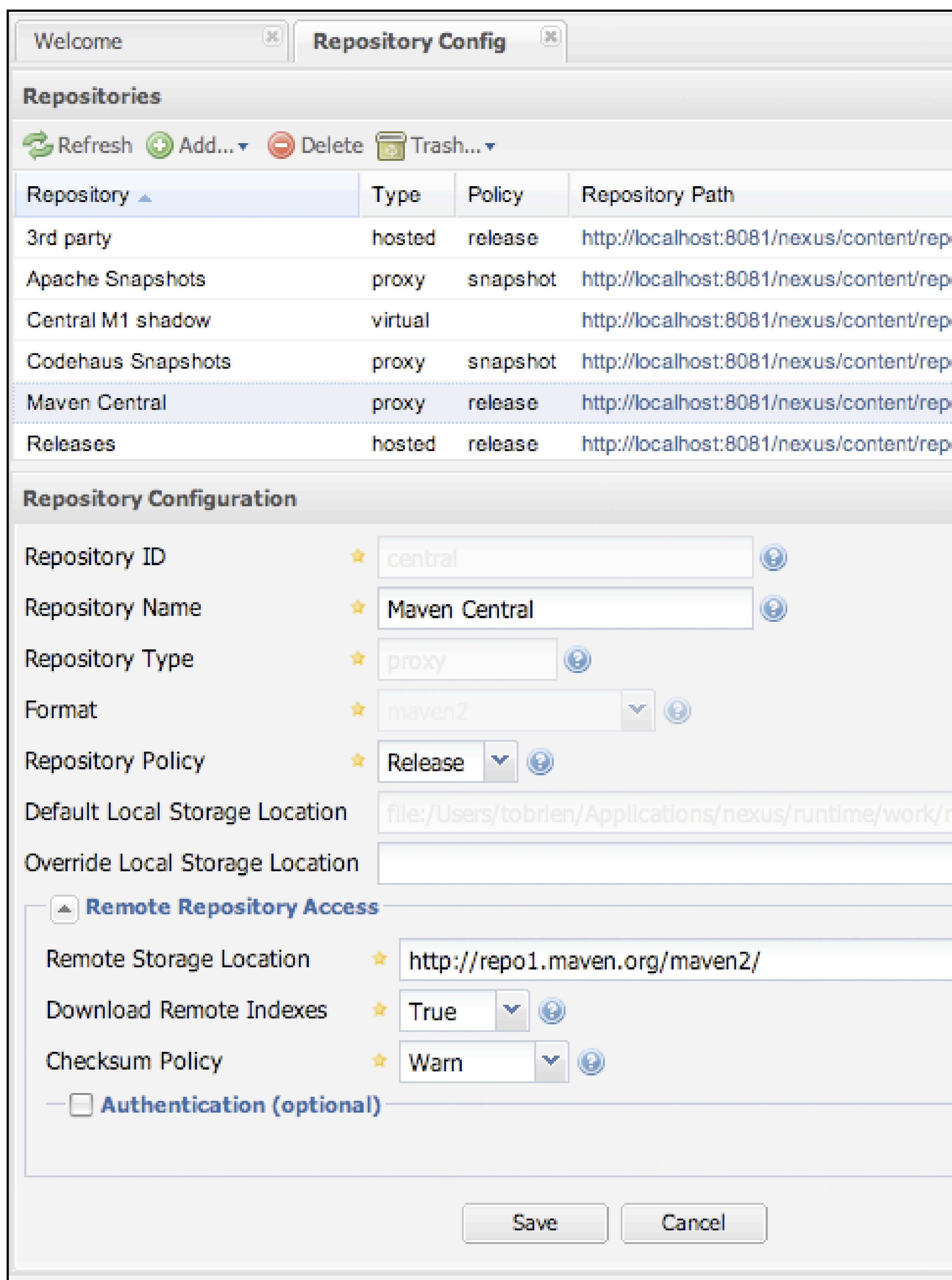


图 16.12. 代理仓库的配置页面

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

图 16.12 “代理仓库的配置页面”展示了Nexus中代理仓库的配置页面。在这个页面中，你可以管理一个外部仓库的设置。本页面中，你可以配置：

仓库ID
: 仓库ID是将会被用在Nexus URL中的标识符。例如，中央代理仓库有一个ID为"central"，这就意味着Maven可以直接在http://localhost:8081/nexus/content/repositories/central访问这个仓库。在一个给定的Nexus安装中，仓库ID必须是唯一的。ID是必需的。

仓库名称
: 仓库的显示名称。名称是必需的。

仓库类型
: 仓库类型（代理，宿主，或者虚拟）。你不能改变仓库的类型，在你创建一个仓库的时候它就被指定了。

仓库策略
: 如果一个代理仓库的策略是release，那么它只会访问远程仓库的发布版本构件。如果一个代理仓库的策略是snapshot，它只会下载远程仓库的快照版本构件。

默认存储位置
: 它不可编辑的，显示出来只是为了参考。这是仓库本地缓存内容的默认存储位置。

覆写存储位置
: 你可以选择为某个特定的仓库覆写存储位置。如果你关心存储空间，或者想要将某个特定仓库（如中央仓库）的内容放到一个不同的位置，你就可以覆写存储位置。

远程仓库访问
: 这一部分告诉Nexus去哪里寻找远程仓库，以及如何与这个被代理的仓库交互。

    远程存储位置
    : 这是远程Maven仓库的URL。

    下载远程索引（本图未显示）
    : 这个字段控制下载远程索引。目前只有中央仓库在http://repo1.maven.org/maven2/.index有一个索引。如果开启它，Nexus会下载这个索引，并使用它用来搜索，以及为任何要求索引的客户（如m2eclipse）服务。新的代理仓库的默认值是开启的，但是Nexus自带的所有仓库的这个默认值是关闭的。要改变Nexus自带的代理仓库设置，更改此选项，保存至仓库，然后给仓库重新编制索引。在这之后，构件搜索会返回中央Maven仓

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

库中可用的每一个构件。第 16.6 节 “维护仓库详细描述了为仓库重新编制索引的过程。

校验和策略

为一个远程仓库设置校验和策略。这个选项默认设置成Warn。该设置可能的值包括：

- Ignore – 完全忽略校验和
- Warn – 如果校验和不正确，在日志中打印一个警告
- StrictIfExists – 如果计算出来的校验和与仓库中的校验和不一致，那就拒绝缓存这个构件。只有校验和文件存在的时候才进行检查。
- Strict – 如果计算出来的校验和与仓库中的校验和不一致，或者如果构件没有校验和文件，就拒绝缓存这个构件。

认证

这一部分允许你为一个远程仓库设置用户名，密码，私钥，密钥口令，NT LAN HOST，以及NT LAN Manager Domain。

访问设置

这一部分为一个仓库配置访问设置。

允许部署

如果允许部署设置成true，Nexus会允许Maven部署构件至这个仓库。允许部署只有对于宿主仓库有意义。

允许文件浏览

如果设置成true，用户可以通过web浏览器来浏览这个仓库的内容。

包含在搜索范围中

如果设置成true，当你在Nexus中执行构件搜索的时候，该仓库会被搜索。如果设置成false，在搜索时该仓库的内容会被排除。

过期失效设置

Nexus为构件和元数据维护一份本地的缓存，你可以为代理仓库配置过期失效参数。过期失效设置有：

未找到缓存TTL

如果Nexus找不到一个构件，它会在一个给定的时间内缓存这个结果。换句话说，如果Nexus不能在远程仓库中找到一个构件，它不会重复的尝试去解析这个构件，除非超过了这个未找到缓存TTL时间。默认值是1140分钟（或者24小时）。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

构件最大年龄

在Nexus从远程仓库获取一个新版本的构件前，告诉它构件的最大年龄是多少。带有release策略的仓库的默认设置是-1，带有snapshot策略的仓库的默认值是1140。

元数据最大年龄

Nexus从远程仓库获取元数据。只有在超过了元数据最大年龄之后，它才会去获取元数据的更新。该设置的默认值是1140分钟（或者24小时）。

HTTP请求设置

这一部分能让你更改对于远程仓库的HTTP请求的属性。该部分中你可以配置请求的User Agent，为请求添加参数，设置超时和重试行为。这一部分涉及的是由Nexus对远程被代理Maven仓库的HTTP请求。

HTTP代理设置

该部分能让你为从Nexus到远程仓库的请求配置HTTP代理。你可以配置一个代理主机，端口和认证设置，以告诉Nexus为所有对远程仓库的请求使用HTTP代理。

16.5.3. 管理组

组是Nexus一个强大的特性，它允许你在一个单独的URL中组合多个仓库。Nexus自带了两个组：public和public-snapshots。public组中组合了三个宿主仓库：3rd Party, Releases, 和Snapshots，还有中央Maven仓库。而public-snapshots组中组合了Apache Snapshots和Codehaus Snapshots仓库。在第 16.4 节 “配置Maven使用Nexus”中我们通过`settings.xml`配置Maven从Nexus管理的public组中寻找构件。图 16.13 “Nexus中的组配置页面”显示了Nexus中的组配置页面，在该图中你可以看到public组的内容。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

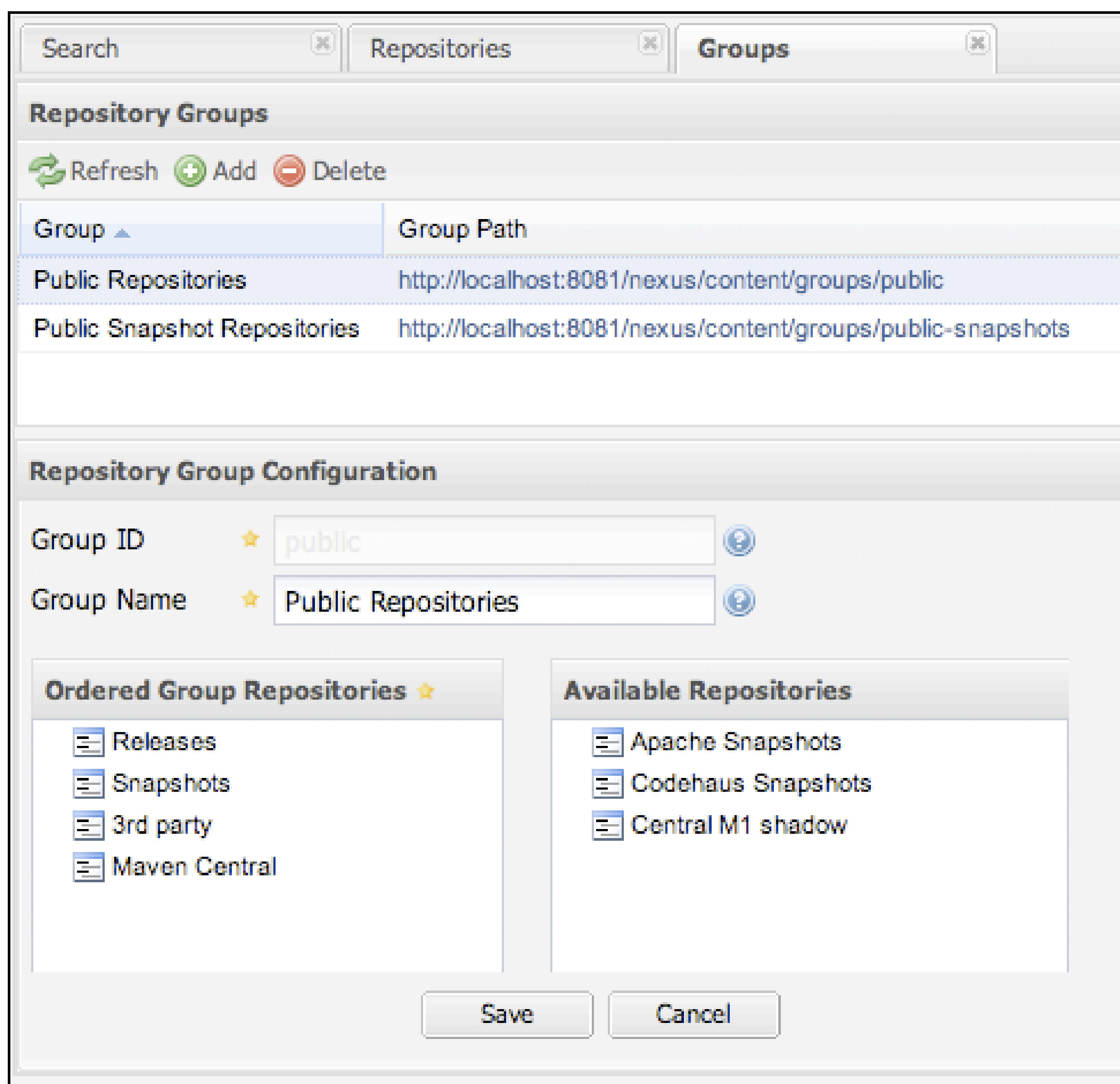


图 16.13. Nexus中的组配置页面

注意在图中有序组仓库中列出的仓库的顺序是很重要的。当Nexus在一组仓库中搜索一个构件的时候，它会返回第一个匹配的结果。要更改列表中仓库的顺序，在有序组仓库选择列表中点击并拖拉仓库即可。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

16.5.4. 管理路由

Nexus 路由就像是你可以应用到Nexus组上的过滤器，当Nexus尝试在一个Nexus组中寻找构件的时候，路由允许你在一个特定的构件搜索中包含或者排除一些仓库。有很多不同的场景中你可能需要配置路由，最平常的是当你想要确信你正从一个特定组的特定的仓库中获取构件。特别是当你想要确信你正从宿主Releases及Snapshots仓库中获取你自己组织的构件的时候，路由功能很有用。当你正尝试从一个Nexus组中解析一个构件的时候，Nexus路由十分适用；当Nexus从一组仓库中解析一个构件的时候，使用路由允许你更改Nexus将查阅的仓库。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

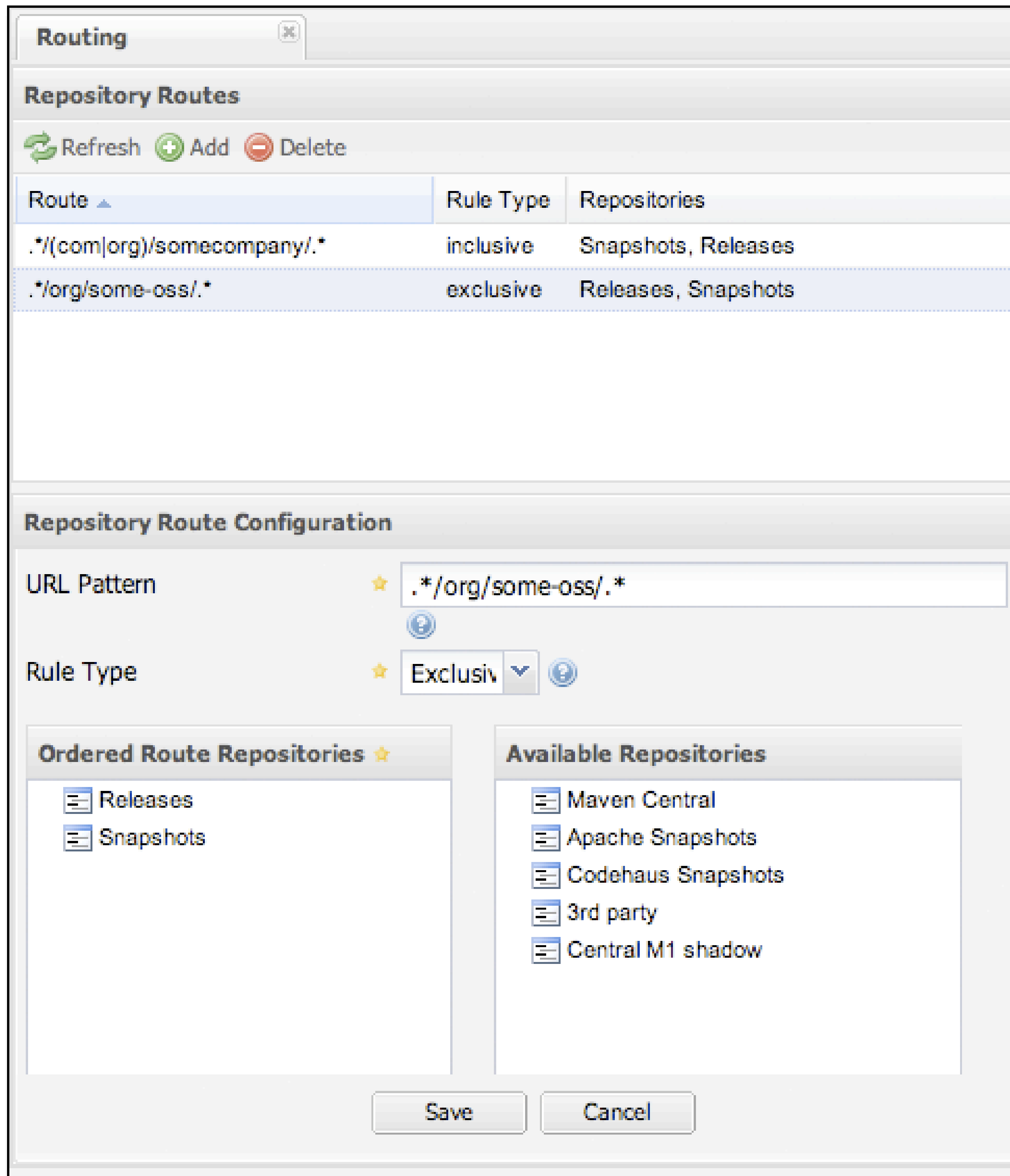


图 16.14. Nexus中的路由配置页面

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

图 16.14 “Nexus中的路由配置页面”显示了路由配置页面。在路由上点击会看到一个页面，能让你配置路由的属性。路由的可用配置选项是：

URL模式

这是Nexus用来匹配请求的模式。如果这个模式与请求表达式匹配，Nexus就会在特定的构件查询中包含或者排除所列出的仓库。在图 16.14 “Nexus中的路由配置页面”中的两个模式是：

.*/(com|org)/somecompany/.*

这个模式会匹配所有包含"/com/somecompany/"或者"/org/somecompany"的路径。圆括弧中的表达式匹配com或者org，"*"匹配一个或多个字符。你可以使用一个像这样的路由来匹配你自己组织的构件，并且将这样的请求对应到宿主的Nexus Releases及Snapshots仓库。

.*/org/some-oss/.*

这个模式用作一个排除路由。它匹配所有包含"/org/some-oss/"的路径。这个特殊的排除路由为所有与该路径匹配的构件排除宿主的Releases和Snapshots仓库。当Nexus尝试解析与该路劲匹配的构件时，它会排除Releases和Snapshots仓库。

路由类型

路由类型可以是“包含”或者“排除”。一个包含路由类型定义了一组仓库，当URL模式匹配的时候这些仓库将被搜索。同样的情况下，一个排除路由定义的仓库则将不会被搜索。

有序路由仓库

这是一个Nexus用来搜索以寻找某个特殊构件的一个有序仓库列表。Nexus从上至下搜索；当它找某个构件的时候，会返回第一个匹配的结果。当它寻找元数据的时候，同一组中所有的仓库会被检查，并且最后返回一个归并的结果。归并的时候，前面的仓库拥有较高的优先权。当一个项目正在寻找LATEST或者RELEASE版本的时候，这可能会影响结果。在一个Nexus组中，你应该在快照版仓库前定义发布版仓库，否则LATEST可能会错误的解析成一个快照版本。

在该图中你你能看到两个Nexus默认带有的假路由。第一个路由是一个包含路由，它是一个自定义路由的例子，一个组织可能用来确信内部生成的构件是从Releases和Snapshots仓库解析的。如果你组织的groupId都由com.somecompany开头，并且你们将内部生成的构件部署到了Releases和Snapshots仓库，该路由让Nexus不浪费时间去从公共的Maven仓库，如中央Maven仓库或者Apache Snapshots仓库，去解析构件。

第二个假路由是一个排除路由。当请求路径包含"/org/some-oss"的时候，路由会排除Releases和Snapshots仓库。如果我们使用"apache"或者"codehaus"替换"some-oss"，这个例子就可能更有意义了。如果这个模式是"/org/apache"，该规则告诉Nexus，当试

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

图解析这些依赖的时候，排除内部的Releases和Snapshots仓库。换句话说，不要浪费时间从你组织的内部仓库中去寻找Apache依赖。

如果两个路由有冲突在怎么办？Nexus会在它处理排除路由之前处理包含路由。记住Nexus路由只会在当搜索一个组的时候影响Nexus解析构件。当Nexus开始从一个Nexus组中解析一个构件，它开始于组中的一个仓库列表。如果有匹配的包含路由，Nexus就会使用组中仓库和包含路由中仓库的交集。在Nexus组中定义的顺序不会受包含路由的影响。Nexus之后就会在这个新的组中应用排除路由。最后在这个结果列表中搜索匹配构件。

总结来说，路由还有很多Nexus的设计者们未预期到的创新可能性，但是，如果你开始信赖冲突或者重叠路由，我们还是建议你小心。保守使用路由，使用教程中的URL模式，随着Nexus的发展，将会有更多的特性允许更细类度的规则以让你阻止特定构件和特定构件版本的请求。记住路由只能用在Nexus组中，当从某个特定仓库中请求一个构件的时候，路由不会被用到。

16.5.5. 网络配置

默认情况下，Nexus监听端口8081。你可以更改这个端口，通过更改`${NEXUS_HOME}/conf/plexus.properties`的值，如例 16.3 “${NEXUS_HOME}/conf/plexus.properties的内容”。为此，停止Nexus，更改文件中`applicationPort`的值，然后重启Nexus。在这之后，你应该能够在`${NEXUS_HOME}/logs/wrapper.log`中看到一条日志记录，告诉你Nexus在监听更改过的端口。

例 16.3. ${NEXUS_HOME}/conf/plexus.properties的内容

```
applicationPort=8081
runtime=/usr/local/hudson/hudson-home/jobs/maven-guide-zh-to-production/workspace/c
apps=${runtime}/apps
work=${runtime}/work
webapp=${runtime}/apps/nexus/webapp
nexus.configuration=${runtime}/apps/nexus/conf/nexus.xml
```

16.6. 维护仓库

在你设置了一系列仓库并且将这些仓库组织成组之后，用户就能够通过点击左边菜单Views部分的Repositories链接，在Neuxs UI上看到一个仓库的列表。Nexus会显示一个仓库列表。这个列表会显示远程仓库的状态；如果要测试一下，可以编辑你的一个仓库，让它拥有一个垃圾远程存储位置URL，你会在仓库管理页面上注意到该仓库的状态变化了。点击一个仓库会显示一个树状视图，以让用户能够浏览仓库的内容。

在一个仓库上右击，会看到一系列能用到仓库上的动作。每个仓库上可用的动作有：

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

查看
: 载入一个仓库的树状视图。该视图能让你深化到特定的目录以查看仓库的内容。

清空缓存
: 为仓库清空缓存。它促使Maven去远程仓库检查更新或者快照版本。它也会重置未找到缓存。

重新编制索引
: 促使Maven为一个仓库重新编制索引。Nexus会重新创建它用来搜索构件请求的索引。如果仓库已被配置了下载远程索引，这一选项促使Nexus从远程仓库下载远程索引。注意如果你开启了远程索引下载，可能需要花一些时间从远程仓库下载索引。当构件搜索结果开始显示没有缓存或请求过的构件，你会知道远程仓库已经更新了。

阻塞代理/允许代理
: 这可以封锁对远程仓库的请求。如果代理被阻塞了，Nexus就不会连接到远程仓库去请求更新。要重新开启远程访问，在仓库上右击然后选择允许代理。当你想要控制代理仓库提供的内容的时候，该选项十分有用。如果你想维护从远程仓库下载内容的严格控制，你可以先基于Nexus运行你组织的构建，然后阻塞所有的代理仓库。

服务失效/服务生效
: 该选项允许你让一个仓库失效，使之不可用。Nexus就会拒绝所有对失效仓库的服务。在你将一个仓库置为不可用之后，你可以通过在一个仓库上右击，选择“服务生效”，来回到可用状态。

16.7. 部署构件至Nexus

不同的组织有不同的理由将构件部署至内部仓库。在有数百（或数千）开发人员的大型组织内，一个内部Maven仓库可以是不同部门之间共享发布版和开发快照版本的有效手段。大部分使用Maven的组织最终都会开始将发布版本和构件部署到一个共享的内部仓库。使用Nexus，可以很容易的部署构件至一个宿主仓库。

要部署构件至Nexus，在`distributionManagement`中提供仓库URL，然后运行mvn deploy。Maven会通过一个简单的HTTP PUT将项目POM和构件推入至你的Nexus安装。最初版本的Nexus没有为宿主仓库提供任何的安全措施。如果你为宿主仓库开启了部署功能，任何人可以连接并部署构件至这个仓库。如果你的Nexus安装能够从公共Internet访问，你绝对会想要将这些仓库的部署功能关闭，或者将你的Nexus安装放到一个如Apache HTTPD的web服务器背后。

你项目的POM不再需要额外的wagon扩展。Nexus可以和Maven内置的`wagon-http-lightweight`一起工作。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

16.7.1. 部署发布版

要部署一个发布版构件至Nexus，你需要配置你项目POM中distributionManagement部分的repository。例 16.4 “为部署配置发布版本仓库”显示了一个发布版部署仓库的样例，这个发布版本仓库的地址是http://localhost:8081/nexus/content/repositories/releases。

例 16.4. 为部署配置发布版本仓库

```
<project>
  ...
  <distributionManagement>
    ...
    <repository>
      <id>releases</id>
      <name>Internal Releases</name>
      <url>http://localhost:8081/nexus/content/repositories/releases</url>
    </repository>
    ...
  </distributionManagement>
  ...
</project>
```

你可以使用你Nexus安装的主机和端口来替换localhost:8081。你的项目有了这个配置之后，你就可以通过执行mvn deploy命令部署构件。

```
$ mvn deploy
[INFO] Scanning for projects...
[INFO] Reactor build order:
[INFO]   Sample Project
[INFO] ------------------------------------------------------------------------
[INFO] Building Sample Project
[INFO]    task-segment: [deploy]
[INFO] ------------------------------------------------------------------------
[INFO] [site:attach-descriptor]
[INFO] [install:install]
[INFO] Installing ~/svnw/sample/pom.xml to ~/.m2/repository/sample/sample\
                                            /1.0/sample-1.0.pom
[INFO] [deploy:deploy]
altDeploymentRepository = null
[INFO] Retrieving previous build number from snapshots
Uploading: http://localhost:8081/nexus/content/repositories/releases/\
           sample/sample/1.0/sample-1.0.pom
24K uploaded
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

注意Nexus可以支持多个宿主仓库；你不需要坚持在默认的releases和snapshots仓库上。你可以为不同的部门创建不同的宿主仓库，然后将多个仓库组合成一个单独的Nexus组。

16.7.2. 部署快照版

要部署快照版本构件至Nexus，你需要配置你项目POM的distributionManagement部分的snapshotRepository。例 16.5 “为部署配置快照版本仓库”显示了快照版本部署仓库的样例，该snapshots仓库配置的地址是http://localhost:8081/nexus/content/repositories/snapshots。

例 16.5. 为部署配置快照版本仓库

```
<project>
  ...
  <distributionManagement>
    ...
    <snapshotRepository>
      <id>Snapshots</id>
      <name>Internal Snapshots</name>
      <url>http://localhost:8081/nexus/content/repositories/snapshots</url>
    </snapshotRepository>
    ...
  </distributionManagement>
  ...
</project>
```

你可以使用你Nexus安装的主机和端口来替换localhost:8081。你的项目有了这个配置之后，你就可以通过执行mvn deploy命令部署构件。如果你项目的版本是快照版本（如1.0-SNAPSHOT）Maven就会将其部署至snapshotRepository：

```
$ mvn deploy
[INFO] Scanning for projects...
[INFO] Reactor build order:
[INFO]   Sample Project
[INFO] ------------------------------------------------------------------------
[INFO] Building Sample Project
[INFO]    task-segment: [deploy]
[INFO] ------------------------------------------------------------------------
[INFO] [site:attach-descriptor]
[INFO] [install:install]
[INFO] Installing ~/svnw/sample/pom.xml to ~/.m2/repository/sample/sample\
                                             /1.0-SNAPSHOT/sample-1.0-20080402.12530
[INFO] [deploy:deploy]
altDeploymentRepository = null
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛


```
[INFO] Retrieving previous build number from snapshots
Uploading: http://localhost:8081/nexus/content/repositories/releases/\
           sample/sample/1.0-SNAPSHOT/sample-1.0-20080402.125302.pom
24K uploaded
```

16.7.3. 部署第三方构件

你的Maven项目可以依赖于一个构件，这个构件不能从中央Maven仓库或任何其它公开Maven仓库找到。有很多原因可能导致这种情形发生：这个构件可能是私有数据库的JDBC驱动如Oracle，或者你依赖于另一个JAR，它既不开源也无法免费获得。在这样的情况下，你就需要手动拿来这些构件然后发布到你自己的仓库中。Nexus提供宿主的"third-party"仓库，就是为了这个目的。

为了阐明发布一个构件至第三方仓库的过程，我们使用一个真实的构件：Oracle JDBC驱动。Oracle发布一个广泛使用的商业数据库产品，该产品带有一个中央Maven仓库没有的JDBC驱动。虽然中央Maven仓库在http://repo1.maven.org/maven2/com/oracle/ojdbc14/10.2.0.3.0/维护了一些Oracle JDBC驱动的POM信息，但这些只是指向Oracle站点的POM。如果你将下列的依赖添加到你的项目。

例 16.6. Oracle JDBC JAR 依赖

```
<project>
  ...
  <dependencies>
    ...
    <dependency>
      <groupId>com.oracle</groupId>
      <artifactId>ojdbc14</artifactId>
      <version>10.2.0.3.0</version>
    </dependency>
    ...
  </dependencies>
  ...
</project>
```

用这个依赖运行一个Maven构建，将会产生如下的输出：

```
$ mvn install
...
[INFO] ------------------------------------------------------------------------
[ERROR] BUILD ERROR
[INFO] ------------------------------------------------------------------------
[INFO] Failed to resolve artifact.

Missing:
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛


```
----------
1) com.oracle:ojdbc14:jar:10.2.0.3.0

  Try downloading the file manually from:
      http://www.oracle.com/technology/software/tech/java/sqlj_jdbc/index.html

  Then, install it using the command:
      mvn install:install-file -DgroupId=com.oracle -DartifactId=ojdbc14 \
               -Dversion=10.2.0.3.0 -Dpackaging=jar -Dfile=/path/to/file

  Alternatively, if you host your own repository you can deploy the file there: \
      mvn deploy:deploy-file -DgroupId=com.oracle -DartifactId=ojdbc14 \
                 -Dversion=10.2.0.3.0 -Dpackaging=jar -Dfile=/path/to/file \
                 -Durl=[url] -DrepositoryId=[id]

  Path to dependency:
   1) sample:sample:jar:1.0-SNAPSHOT
   2) com.oracle:ojdbc14:jar:10.2.0.3.0

----------
1 required artifact is missing.
```

Maven构建失败了因为它不能在Maven仓库中找到Oracle JDBC驱动。要补救这种情况，你将需要发布Oracle JDBC构件至你的Nexus third-party仓库。为此，从http://www.oracle.com/technology/software/tech/java/sqlj_jdbc/index.html下载Oracle JDBC驱动，然后保存至文件`ojdbc.jar`。使用以下命令发布该文件至Nexus：

```
$ mvn deploy:deploy-file -DgroupId=com.oracle -DartifactId=ojdbc14 \
> -Dversion=10.2.0.3.0 -Dpackaging=jar -Dfile=ojdbc.jar \
> -Durl=http://localhost:8081/nexus/content/repositories/thirdparty \
> -DrepositoryId=thirdparty
...
[INFO] [deploy:deploy-file]
Uploading: http://localhost:8081/nexus/content/repositories/thirdparty/\
                  com/oracle/ojdbc14/10.2.0.3.0/ojdbc14-10.2.0.3.0.jar
330K uploaded
[INFO] Retrieving previous metadata from thirdparty
[INFO] Uploading repository metadata for: 'artifact com.oracle:ojdbc14'
[INFO] Retrieving previous metadata from thirdparty
[INFO] Uploading project information for ojdbc14 10.2.0.3.0
```

在你运行mvn deploy:deploy-file之后，该构件会被发布至Nexus的third-party仓库。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

第 17 章 编写插件

17.1. 简介

虽然本章介绍的是高级话题,关于编写Maven插件，但不要被吓到。当对于其所有理论和复杂性来说，基本的概念还是比较容易理解的，而编写插件的方法也十分易懂。在你阅读本章之后，你将能更好的掌握编写Maven插件所涉及的内容。

17.2. Maven编程

本书的绝大部分都在讲述使用Maven，在这本关于Maven的书中，你还没看到太多关于定制Maven的样例代码。事实上，到目前为止这样的代码还没有。这原是设计的目的，99%的Maven用户将从来不需要编写自定义插件来定制Maven；有大量的可配置的插件可用，除非你有特别的单独需求，你没有理由自己去编写一个新的插件。而在那一小部分编写自定义插件的人中，也只有更少的人需要去查看Maven的源码以自定义Maven核心组件。如果你确实需要自定义Maven的行为，那么你可以编写一个插件。修改核心Maven代码与修改操作系统的TCP/IP堆栈一样，远远超出了大部分开发人员的范围，这对于大部分Maven用户来说过于抽象。

另一方面，如果你想开始编写自定义插件，你就需要学习一些Maven的内部原理：它如何管理软件组件？什么是一个插件？我如何能够自定义生命周期？本节将解答其中的一些问题，并介绍一些Maven设计核心的概念。学习如何编写Maven插件同时也是学习定制Maven本身一种很好的方式。如果你想了解如何开始探究Maven背后的代码，那么你已经找到了正确的出发点。

17.2.1. 什么是反转控制？

Maven的心脏是一个名为Plexus的反转控制（IoC）框架。它做什么呢？这是一个用来管理及关联组件的系统。虽然有一篇Martin Fowler编写的关于Ioc的权威论文，但这个概念和术语在过去的一些年中已经被大量的重载，因此找出一个关于此概念的良好定义已经变得非常困难，很多定义都是自我指认的（或者对于上述论文的简单引用）。这里我们将通过一个比喻来简要介绍反转控制和依赖注入，而不是简单的援引维基百科。

假设你有一系列的组件需要装配在一起。当你考虑组件的时候，将其想象成立体声音响组件而非软件组件。想象一下有一些音响组件接通了一个Playstation 3和一个机顶盒，后者又需要连接一个苹果电视盒以及一个50英寸的平板液晶电视。你从电子产品商店将所有这些东西带回家，并购买了一些电缆准备用来连接这些电器。你将所有组件拆开，将其放到正确的位置，然后开始连接同轴电缆，数字输入设备，立体声控制设备，以及网线等等。从你的家庭娱乐中心后退一步，然后打开电视机，你已经完成了依赖注入，你刚才正处在一个反转控制容器中。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

那么，你到底需要做什么？你的Playstation 3和一个Java Bean都提供接口。Playstation 3有两个输入：电源和网络，以及一个电视输出。你的Java Bean有三个属性：power，network，和tvOutput。当你打开Playstation 3盒子的时候，它并不会提供关于连接所有不同种类电视的详细图片和指令，当你查看你的Java Bean的时候，它也只是提供了一组属性，而不是创建和管理整个系统组件的显式的方法。在一个如Plexus的IoC容器中，你负责通过简单的提供输入输出接口来声明组件之间的关系。你不需要初始化对象，Plexus会帮你完成；你的应用程序的代码不用去管理组件的状态，Plexus负责。虽然这听起来非常俗套，不过这里还是要说，当你启动Maven的时候，实际上是启动Plexus来管理一个带有很多相互关联组件的系统，就像你的家庭影音系统一样。

那么，使用Ioc容器的优点是什么呢？购买离散的音响组件的优点是什么？如果一个组件坏了，你可以放一个Playstation 3的替代品，而不用花20,000美元去重新购买整个系统。如果你对电视机不满意，你可以在不影响CD播放器的情况下将电视换掉。对你来说最重要的是，你的音响组件现在花费更低，但功能更强，且更稳定，因为制造商现在根据一组通用的输入输出来制造组件，他们能够更加的关注组件本身。反转控制和依赖注入提倡分解以及标准的形成。软件产业喜欢自己给新的想法命名，但依赖注入和反转控制实际上只是对于分解及可交换体系的重新命名。如果你真的想要好好了解DI和IoC，你可以学习一下Model T，Cotton Gin，以及19世纪末期形成的一个铁路系统标准。

17.2.2. Plexus简介

用Java实现的IoC容器中，最重要的一个功能是称作依赖注入的机制。IoC的基本想法是将对象的创建和管理从代码中剥离，并将控制放到IoC框架手中。在一个面向接口编程的应用程序中使用依赖注入，你创建的组件可以不与任何特定的接口实现绑定。你的程序也针对接口编程并通过配置Plexus来将正确的实现连接到正确的组件。虽然你的代码都是与接口打交道，但你仍然可以通过一个定义组件的XML文件来获得类和组件相互依赖的信息。换句话说，你可以编写独立的组件，然后你可以通过一个XML文件来定义组件应当如何被装配在一起。在Plexus的情形中，定义系统组件的XML文档位于`META-INF/plexus/components.xml`。

在一个Java IoC容器中，有很多中方法将依赖值注入到一个组件对象中：构造器，set方法，或者字段注入。虽然Plexus提供全部这三种依赖注入技术，Maven只使用其中的两种：字段注入和set方法注入。

构造器注入
: 构造器注入是指当对象实例被创建的时候，通过对象的构造方法填入对象的值。例如，如果你有类对象`Person`，其构造方法是`Person(String name, Job job)`，你就可以通过该构造器传入`name`和`job`的值。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

set方法注入
: set方法注入是指使用Java Bean属性的set方法来填入对象依赖。例如，如果你有一个带有name和job属性的Person对象，一个使用set方法注入的IoC容器会使用无参构造器创建一个Person的实例，之后，它会继续调用setName()和setJob()方法。

字段注入
: 构造器和set方法注入都依赖于公共方法。而使用字段注入的时候，IoC容器通过直接设置对象字段的值来填入组件依赖。例如，如果你有一个带有name和job字段的Person对象，你的IoC容器会直接设置这两个字段来填入依赖（如：person.name = "Thomas"; person.job = job;）。

17.2.3. 为什么使用Plexus?

目前Spring是最流行的IoC容器。它影响了Java的“生态系统”，迫使如Sun Microsystems之类的公司让出更多的对于开源社区的控制，并通过提供一个更易插入的，面向组件的“总线”来帮助开发一些标准。但是Spring不是唯一的开源IoC容器。事实上有很多IoC容器（如PicoContainer[1]）。

很多年以前，当Maven被创建的时候，Spring并不是一个成熟的选项。最初的Maven提交者团队更熟悉Plexus因为正是他们发明了Plexus，因此他们决定使用Plexus做为Ioc容器。虽然它没有Spring Framework流行，但并不是说它的功能没那么强大。而且，事实上它是由创建Maven的同一个人发明的，这使其更适合Maven。阅读本章之后你会了解Plexus如何工作。如果你使用过一个IoC容器你会看到Plexus和你使用的另外一个IoC容器的相似性和差异性。

注意

Maven基于Plexus并不意味着Maven社区是“反Spring”的（我们在本书中包含了整整的一章关于Spring样例的内容，Spring项目的一部分也转移到Maven作为构建平台）。由于“你们为什么不用Spring”这个问题经常出现，因此在这里解释这个问题。我们知道，Spring是一个明星，我们并不拒绝它，但为人们介绍Plexus仍然是我们持续要做的工作：软件产业中，更多的选择总是好事。

17.2.4. 什么是插件？

一个Maven插件是包含了一个插件描述符和一个或者多个Mojo的Maven构件。一个Mojo可以被认为是Maven中的一个目标，每一个目标对应了一个Mojo。compiler:compile目标对应了Maven Compiler插件的CompilerMojo类，jar:jar目标对应了Maven Jar插件

[1] http://www.picocontainer.org/

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

的JarMojo类。当你编写自己的插件的时候，你在一个单独的插件构件中将一组相互关联的Mojo（或者目标）归类。

2

注意

Mojo?什么是Mojo?词mojo[2]被定义为“一种神奇的魔力或咒语”，“一个护身符，通常是一个包含了一个或多个神奇物件的法兰绒小包”，以及“个人魅力；吸引力”。Maven使用术语Mojo因为这是一个对于Pojo（Plan-old Java Object）的玩笑。

Mojo不仅仅是Maven中的一个目标，它是一个由Plexus管理的组件，可以引用其它Plexus组件。

17.3. 插件描述符

Maven插件包含了一个告诉Maven各种Mojo和插件配置的路线图。这就是插件描述符，它位于JAR文件中的META-INF/maven/plugin.xml。当Maven载入一个插件的时候，它读取该XML文件，初始化并配置插件对象，使Mojo被包含在插件中，供Maven使用。

当你编写自定义Maven插件的时候，你几乎不需要编写插件描述符。在第 10 章构建生命周期中，绑定到maven-plugin打包类型的生命周期目标显示，plugin:descriptor目标被绑定到了generate-resources生命周期阶段。该目标根据插件源码中的注解生成一个插件描述符。本章后面，你会看到如何注解Mojo，并且你会了解这些注解最终如何成为META-INF/maven/plugin.xml文件的内容。

例 17.1 “插件描述符”展示了Maven Zip插件的描述符。该插件简单的对输出目录进行zip压缩并归档。一般来说，你不需要编写自定义插件从Maven创建归档，你可以使用Maven Assembly插件，该插件能够以多种格式帮助生成分发归档。仔细阅读下面的插件描述符以了解其包含的内容：

[2] “mojo.”美国传统英语字典，第四版。Houghton Mifflin公司，2004，Answer.com 02 Mar. 2008. http://www.answers.com/topic/mojo-1

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

```
  <mojos>
    <mojo>
      <goal>zip</goal>
      <description>Zips up the output directory.</description>
      <requiresDirectInvocation>false</requiresDirectInvocation>
      <requiresProject>true</requiresProject>
      <requiresReports>false</requiresReports>
      <aggregator>false</aggregator>
      <requiresOnline>false</requiresOnline>
      <inheritedByDefault>true</inheritedByDefault>
      <phase>package</phase>
      <implementation>com.training.plugins.ZipMojo</implementation>
      <language>java</language>
      <instantiationStrategy>per-lookup</instantiationStrategy>
      <executionStrategy>once-per-session</executionStrategy>
      <parameters>
        <parameter>
          <name>baseDirectory</name>
          <type>java.io.File</type>
          <required>false</required>
          <editable>true</editable>
          <description>Base directory of the project.</description>
        </parameter>
        <parameter>
          <name>buildDirectory</name>
          <type>java.io.File</type>
          <required>false</required>
          <editable>true</editable>
          <description>Directory containing the build files.</description>
        </parameter>
      </parameters>
      <configuration>
        <buildDirectory implementation="java.io.File">/usr/local/hudson/hudson-home
        <baseDirectory implementation="java.io.File">/usr/local/hudson/hudson-home/
      </configuration>
      <requirements>
        <requirement>
          <role>org.codehaus.plexus.archiver.Archiver</role>
          <role-hint>zip</role-hint>
          <field-name>zipArchiver</field-name>
        </requirement>
      </requirements>
    </mojo>
  </mojos>
  <dependencies>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-io</artifactId>
    <version>1.3.2</version>
  </dependencies>
</plugin>
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

该插件描述符有三个部分：插件的顶层配置，包含如groupId和artifactId之类的元素；mojo声明；以及依赖声明。让我们仔细解释一下每一部分。

17.3.1. 顶层插件描述符元素

plugin元素中顶层的配置元素有：

description
: 该元素包含了插件的简短描述。在Zip插件的情况中，该描述为空。

groupId, artifactId, version
: 就像Maven中的任何其它构件一样，插件也需要唯一的坐标。groupId, artifactId,和version用来在Maven仓库中定位插件。

goalPrefix
: 该元素（目标前缀）用来设置某个特定插件用来引用目标的前缀。如果你看一下Compiler插件的描述符，你会看到goalPrefix的值为compile，如果你看一下Jar插件的描述符，你会看到goalPrefix为jar。为自定义插件选择一个独一无二的前缀非常重要。

isolatedRealm (不赞成使用)
: 该遗留属性不再被Maven使用。它的存在是为了给旧的插件提供向后兼容性。早期版本的Maven使用它提供一种在单独ClassLoader中载入插件依赖的机制。Maven扩展使用了Codehaus[3]社区中一个名为ClassWorlds[4]的项目，创建由ClassRealm对象建模的ClassLoader对象层次结构。尽管忽略该属性，永远将其设置成false。

inheritedByDefault
: 如果inheritedByDefault（缺省继承）被设置成true，所有在父项目配置的该插件的mojo会在子项目中生效。如果你配置一个mojo在父项目中特定的阶段执行，并且该插件inheritedByDefault属性的值为true，这段执行会被子项目继承。如果inheritedByDefault没有被设置成true，那么定义在父项目中的目标执行不会被子项目继承。

17.3.2. Mojo配置

接下来是每个Mojo的声明。plugin元素包含一个名为mojos的元素，它为每个插件中的mojo元素包含一个mojo元素。每个mojo元素包含如下的配置元素：

[3] http://www.codehaus.org
[4] http://classworlds.codehaus.org/

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

goal
: 这是目标的名称。如果你在运行compiler:compile目标，compiler就是插件的goalPrefix，compile就是目标的名称。

description
: 目标的简要描述，当用户使用Help插件生成插件文档的时候，该描述会被显示。

requiresDirectInvocation
: 如果你将其设置成true，那么该目标就只能由用户在命令行显示的执行。如果有人想要将该目标绑定到一个生命周期阶段，Maven会打印错误信息。该元素默认值是false。

requiresProject
: 指定该目标不能在项目外部运行。目标需要一个带有POM的项目。requiresProject默认的值为true。

requiresReports
: 如果你正创建一个插件，它依赖于报告，那么你就需要将requiresReports设置成true。例如，如果你创建一个插件用来聚合许多报告的信息，那么就需要将requiresReports设置成true。该元素默认的值为false。

aggregator
: 当Mojo描述符的aggregator设置成true的时候，那么该目标只会在Maven执行的时候运行一次，提供该配置是为了让开发人员能够对一系列构建进行总结；例如，创建一个插件来概述构建中所有项目的一类报告。一个aggregator设置成true的目标应该只在Maven构建的顶层项目中运行。aggregator默认值是false。aggregator在未来版本的Maven中很有可能被弃用。

requiresOnline
: 指定当Maven在离线模式（-o命令行选项）的时候该目标不能运行。如果一个目标依赖于网络资源，你就需要将该元素设置成true，那么如果在离线模式下运行，Maven就会输出错误信息。该元素默认值是false。

inheritedByDefault
: 如果inheritedByDefault被设置成true，在父项目中配置的mojo就会同样在子项目中生效。如果你配置一个mojo在父项目中特定的阶段执行，并且该插件inheritedByDefault属性的值为true，这段执行会被子项目继承。如果inheritedByDefault没有被设置成true，那么定义在父项目中的目标执行不会被子项目继承。

phase
: 如果用户没有为该目标绑定一个阶段，那么该元素定义一个mojo默认的阶段。如果你没有不指定phase元素，Maven就会要求用户在POM中显式的指定一个阶段。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

implementation
: 该元素告诉Maven需要为该Mojo初始化什么类。这是一个Plexus组件属性（在Plexus ComponentDescriptor 中定义）。

language
: Maven Mojo默认的语言是java。该配置控制Plexus ComponentFactory初始化该Mojo组件。本章关注于使用Java编写Maven插件，但是你也可以使用其它很多语言来编写Maven插件，如Groovy, Beanshell,和Ruby。如果你使用这其中的一种语言来编写插件，那么就需要设置language元素的值。

instantiationStrategy
: 该属性是一个Plexus组件配置属性，它告诉Plexus如何创建和管理组件实例。在Maven中，所有mojo的instantiationStrategy都被配置成per-lookup，每次Maven从Plexus获取该mojo的时候，一个新的实例被创建。

executionStrategy
: executionStrategy告诉Maven什么时候，怎样运行一个Mojo。可用的值是once-per-session和always。老实说，任何值都是可用的，这个特殊的属性并不做什么事情，它是从早期Maven设计中遗留下来的。在未来版本的Maven中该属性很有可能被弃用。

parameters
: 该元素描述Mojo的所有参数。参数名称是什么？参数类型是什么？是否是必须的？每个参数拥有如下的元素：

  name
  : 参数名（如 baseDirectory）

  type
  : 参数类型（Java类）（如java.io.File）

  required
  : 参数是否是必须的？如果为true，那么当目标运行的时候该参数不能为null。

  editable
  : 如果一个参数不是可编辑的（如果editable被设置成false），那么该参数的值就不能在POM中设置。例如，如果插件描述符定义了buildDirectory的值为/usr/local/hudson/hudson-home/jobs/maven-guide-zh-to-production/workspace/content-zh，那么在POM中该值就不能被重写。

  description
  : 当生成插件文档的时候（使用Help插件），该插件的简短描述。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

configuration
: 该元素为所有Mojo参数提供默认值。本例中为Mojo参数`baseDir`和`buildDirectory`提供了默认值，这里，属性implementation指定了参数的类型（本例中是`java.io.File`），而其元素值包含了一个硬编码的默认值，或者一个Maven属性引用。

requirements
: 这是一个描述符十分有趣的地方。一个mojo是一个由Plexus管理的组件，而且，由于该原因，它就有机会引用Plexus管理的其它组件。该元素能让你定义对于其它Plexus组件的依赖。

虽然你的确应该能够读懂插件描述符，但你几乎从不需要去手工的编写一个描述符文件。插件描述符会根据Mojo的一组注解自动的生成。

17.3.3. 插件依赖

最后，插件描述符像Mave项目一样声明了一组依赖。当Maven使用插件的时候，它会在运行插件之前之前下载所有需要的依赖。在本例中，该插件依赖于Jakarta Commons IO版本 1.3.2 。

17.4. 编写自定义插件

在你编写自定义插件的时候，你实际上是要编写一系列的Mojo（目标）。每个Mojo是一个单独的Java类，它包含了一系列注解来告诉Maven如何生成插件描述符。在编写Mojo类之前，你需要使用正确的打包类型和POM来创建一个Maven项目。

17.4.1. 创建一个插件项目

要创建一个插件项目，你应该使用Maven Archetype插件。以下的命令会创建一个插件，其`groupId`是`org.sonatype.mavenbook.plugins`，`artifactId`是`first-maven-plugin`:

```
$ mvn archetype:create \
  -DgroupId=org.sonatype.mavenbook.plugins \
  -DartifactId=first-maven-plugin \
  -DarchetypeGroupId=org.apache.maven.archetypes \
  -DarchetypeArtifactId=maven-archetype-mojo
```

Archetype插件会创建一个名为my-first-plugin的目录，其包含了如下的POM:

例 17.2. 一个插件项目的POM

```
<?xml version="1.0" encoding="UTF-8"?><project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.sonatype.mavenbook.plugins</groupId>
  <artifactId>first-maven-plugin</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>maven-plugin</packaging>
  <name>first-maven-plugin Maven Mojo</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>org.apache.maven</groupId>
      <artifactId>maven-plugin-api</artifactId>
      <version>2.0</version>
    </dependency>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

在一个插件项目的POM中最重要的元素是打包类型，其值为maven-plugin。该打包类型会定制Maven的生命周期，使其包含创建插件描述符必要的目标。插件生命周期在第 10.2.3 节 “Maven Plugin”中介绍，它和Jar生命周期类似，但有三个例外：plugin:descriptor被绑定到generate-resources阶段，plugin:addPluginArtifactMetadata被添加到package阶段，plugin:updateRegistry被添加到install阶段。

插件项目的POM中另一个重要的部分是，它有一个对于Maven Plugin API的依赖。该项目依赖于maven-plugin-api的2.0版本，同时它也有一个测试范围的JUnit依赖。

17.4.2. 一个简单的Java Mojo

本章，我们将介绍一个用Java编写的Maven Mojo。你项目中每一个Mojo都要实现org.apache.maven.plugin.Mojo接口，下例中的Mojo通过扩展org.apache.maven.plugin.AbstractMojo类实现了该接口。在我们深入Mojo的代码之前，让我们花一些时间看一下Mojo接口。Mojo提供过了如下的方法：

`void setLog( org.apache.maven.monitor.logging.Log log )`
: 每一个Mojo实现都必须提供一种方法让插件能够和某个特定目标的过程相交流。该目标成功了么？或者，是否在运行目标的时候遇到了问题？当Maven加载并运

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

行Mojo的时候，它会调用setLog()方法，为Mojo实例提供正确的日志目标，以让你在自定义插件中使用。

protected Log getLog()
: Maven会在Mojo运行之前调用setLog()方法，然后你的Mojo就可以通过调用getLog()获得日志对象。你的Mojo应该去调用这个Log对象的方法，而不是直接将输出打印到标准输出或者控制台。

void execute() throws org.apache.maven.plugin.MojoExecutionException
: 轮到运行你目标的时候，Maven就会调用该方法。

Mojo接口只关心两件事情：目标运行结果的日志记录，以及运行一个目标。当你编写自定义插件的时候，你会要扩展AbstractMojo。AbstractMojo处理setLog()和getLog()的实现，并包含一个抽象的execute()方法。在你扩展AbstractMojo的时候，你所需要做的只是实现execute()方法。例 17.3 “一个简单的EchoMojo”展示了一个简单的Mojo实现，它只是打印一条简单的信息到控制台。

例 17.3. 一个简单的EchoMojo

```
package org.sonatype.mavenbook.plugins;

import org.apache.maven.plugin.AbstractMojo;
import org.apache.maven.plugin.MojoExecutionException;
import org.apache.maven.plugin.MojoFailureException;

/**
 * Echos an object string to the output screen.
 * @goal echo
 */
public class EchoMojo extends AbstractMojo
{
    /**
     * Any Object to print out.
     * @parameter expression="${echo.message}" default-value="Hello Maven World..."
     */
    private Object message;

    public void execute()
        throws MojoExecutionException, MojoFailureException
    {
        getLog().info( message.toString() );
    }
}
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

如果你在前一节中创建的项目的/usr/local/hudson/hudson-home/jobs/maven-guide-zh-to-production/workspace/content-zh下，按照路径src/main/java/org/sonatype/mavenbook/mojo/EchoMojo.java创建该Mojo，然后运行mvn install，你就可以在命令行直接调用该目标。

```
$ mvn org.sonatype.mavenbook.plugins:first-maven-plugin:1.0-SNAPSHOT:echo
```

这一长串命令行是mvn后面加上groupId:artifactId:version:goal。在你运行该命令之后你应该会看到一个包含了目标echo信息的输出："Hello Maven World"。如果你想要自定义该信息，你可以如下在命令行传入信息参数：

```
$ mvn org.sonatype.mavenbook.plugins:first-maven-plugin:1.0-SNAPSHOT:echo \
             -Decho.message="The Eagle has Landed"
```

仍然按照前一条命令运行EchoMojo，会得到这样的输出："The Eagle has Landed"。

17.4.3. 配置插件前缀

在命令行声明groupId，artifactId，version和goal十分麻烦。为了处理这个问题，Maven为插件分配了前缀。你可以使用插件前缀jar，然后使用命令mvn jar:jar，而非：

```
$ mvn org.apache.maven.plugins:maven-jar-plugin:2.2:jar
```

Maven是如何解析jar:jar至org.apache.mven.plugins:maven-jar:2.3的呢？Maven查看Maven仓库中的一个文件然后获得一些列含有特定groupId的插件。默认情况下，Maven被配置成从两个组寻找插件：org.apache.maven.plugins和org.codehaus.mojo。当你指定一个新的插件前缀如mvn hibernate3:hbm2ddl的时候，Maven会为了正确的插件前缀扫描仓库元数据。首先，Maven会扫描org.apache.maven.plugins组来查找插件前缀hibernate3。如果它没有在这个组中找到该插件前缀，它就会接着扫描org.codehaus.mojo组的元数据。

当Maven针对某个特定的groupId扫描元数据的时候，它从Maven仓库获取一个XML文件，该文件包含了这个组中构件的元数据。该XML文件对于每个遵循参考实现的仓库来说都是明确的，如果你不在使用一个自定义仓库，你就能在你的本地Maven仓库（~/.m2/repository）路径org/apache/maven/plugins/maven-metadata-central.xml下看到组org.apache.maven.plugins的Maven元数据。例 17.4 "Maven插件组的Maven元数据"展示了这样一个XML文件的代码片段。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

例　17.4.　Maven插件组的Maven元数据

```
<?xml version="1.0" encoding="UTF-8"?>
<metadata>
  <plugins>
    <plugin>
      <name>Maven Clean Plugin</name>
      <prefix>clean</prefix>
      <artifactId>maven-clean-plugin</artifactId>
    </plugin>
    <plugin>
      <name>Maven Compiler Plugin</name>
      <prefix>compiler</prefix>
      <artifactId>maven-compiler-plugin</artifactId>
    </plugin>
    <plugin>
      <name>Maven Surefire Plugin</name>
      <prefix>surefire</prefix>
      <artifactId>maven-surefire-plugin</artifactId>
    </plugin>
    ...
  </plugins>
</metadata>
```

正如你在例　17.4 “Maven插件组的Maven元数据”中所看到的，你本地仓库的maven-metadata-central.xml文件能帮助你运行mvn surefire:test。Maven扫描org.apache.maven.plugins和org.codehaus.mojo: org.apache.maven.plugins中的插件被认为是核心Maven插件，而org.codehaus.mojo中的被认为是额外的插件。Apache Maven项目管理org.apache.maven.plugins组，而另外一个独立的开源社区管理Codehaus Mojo项目。如果你想要发布插件到你自己的groupId下，你就需要让Maven自动扫描你的groupId以获得插件前缀，你就可以通过Maven settings自定义Maven需要扫描的插件组。

如果你想要通过first:echo命令就能运行first-maven-plugin的echo目标，如所示，添加org.sonatype.mavenbook.plugins groupId至你的~/.m2/settings.xml文件中。这会让Maven优先扫描org.sonatype.mavenbook.plugins插件组。

例　17.5.　在Maven Settings中自定义插件组

```
<settings>
  ...
  <pluginGroups>
    <pluginGroup>org.sonatype.mavenbook.plugins</pluginGroupd>
  </pluginGroups>
</settings>
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

你可以在任何目录运行mvn first:echo，并看到Maven正确解析了目标前缀。这样子行得通是因为我们遵循了Maven插件的命名约定。如果你的插件有一个artifactId，并且它遵循模式maven-first-plugin，或者first-maven-plugin。Maven就会自动为你的插件赋予前缀first。换句话说，当Maven Plugin插件为你的插件生成插件描述符的时候，你不需要显式的为你的项目设定goalPrefix，###artifactId##########plugin:descriptor目标会从你插件的artifactId中抽取前缀。

- ${prefix}-maven-plugin, OR
- maven-${prefix}-plugin

如果你想要显式的设定插件前缀，你需要配置Maven Plugin插件。Maven Plugin插件被用来构建插件描述符，并在打包和加载阶段运行一些插件特定的任务。Maven Plugin插件可以像其它任何插件一样在build元素下配置。要为你的插件设置插件前缀，在项目first-maven-plugin的pom.xml中添加如下的build元素：

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

例 17.6. Configuring a Plugin Prefix

```
<?xml version="1.0" encoding="UTF-8"?><project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.sonatype.mavenbook.plugins</groupId>
  <artifactId>first-maven-plugin</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>maven-plugin</packaging>
  <name>first-maven-plugin Maven Mojo</name>
  <url>http://maven.apache.org</url>
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-plugin-plugin</artifactId>
        <version>2.3</version>
        <configuration>
          <goalPrefix>blah</goalPrefix>
        </configuration>
      </plugin>
    </plugins>
  </build>
  <dependencies>
    <dependency>
      <groupId>org.apache.maven</groupId>
      <artifactId>maven-plugin-api</artifactId>
      <version>2.0</version>
    </dependency>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

例 17.6 “Configuring a Plugin Prefix”设置了插件前缀为`blah`。如果在`~/.m2/settings.xml`中的`pluginGroups`下添加了`org.sonatype.mavenbook.plugins`元素，你就应该能够在任何目录通过mvn blah：echo命令运行`EchoMojo`。

17.4.4. 插件中的日志

Maven通过在运行Mojo之前调用`setLog()`来连接你的Mojo至日志提供程序。该提供程序提供了一个`org.apache.maven.monitor.logging.Log`的实现。该类暴露了一些你可以用

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

来和用户交流信息的方法。这个Log类提供了很多日志级别，与Log4J[5]提供的API十分类似。每个级别：debug，info，error，warn，都有一系列可用的方法。为了节省时间，我们只列出了debug级别的方法：

`void debug( CharSequence message )`
: 打印信息至debug日志级别。

`void debug( CharSequence message, Throwable t )`
: 打印信息至debug日志级别，并包含一个Throwable（Exception或者Error）的堆栈信息。

`void debug( Throwable t )`
: 打印一个Throwable（Exception或者Error）的堆栈信息。

每个级别暴露同样的一组方法。这四个日志级别服务于不同的目的。debug级别是为了调试目的，人们用来观察mojo运行的详细情况。你应该使用debug日志级别来提供Mojo运行尽可能详细的细节，但你从不应该去假设用户会看到debug级别信息。info级别是为了生成一些正常操作的信息。如果你构建一个插件使用编译器来编译代码，你可能就需要打印编译的输出至屏幕。

warn日志级别用来记录一些非预期的，但你的Mojo还能够处理的事件和错误。如果你试图运行编译Ruby源码的插件，而实际上没有可用的Ruby源码，你就需要打印一个警告信息，然后继续运行。警告（warn）不是致命的，但是错误（error）就是一些终止程序运行的情况。对于那些完全非预期的错误情况，这里有error日志级别。如果你不再能够继续运行一个Mojo，你就需要使用error。如果你在编写一个Mojo编译Java源码，但编译器不可用，你就需要打印一条信息到error级别，然后传递一条异常，以让Maven能够输出给用户。你应该假设用户能够看到大多数的info信息，以及所有的error信息。

17.4.5. Mojo类注解

在first-maven-plugin中，我们并没有编写插件描述符，我们依赖于Maven从源码生成插件描述符。该描述符根据你插件项目的POM信息以及EchoMojo类上一系列的注解生成。EchoMojo仅仅声明了一个@goal注解，这里有一个注解的列表，你可以将其用到Mojo实现上。

@goal <goalName>
: 这是唯一必需的注解，它给予目标一个在插件中唯一的名称。

@requiresDependencyResolution <requireScope>
: 标记该mojo在可以运行之前，需要特定范围（或者一个暗指的范围）的依赖。支持的范围有compile，runtime，和test。如果该注解有一个值为test，那么就是告诉Maven，除非测试范围的所有依赖都被正确解析了，否则该mojo不能运行。

[5] http://logging.apache.org/

@requiresProject (true|false)
标记该mojo必须在一个项目中运行，默认为true。这一点插件类型和骨架类型（archetype）相反，后者默认为false。

@requiresReports (true|false)
如果你正创建一个依赖于报告的项目，你就需要将requiresReports设置成true。该注解默认的值是false。

@aggregator (true|false)
一个aggregator设置成true的Mojo在Maven运行的时候只会被执行一次，有了该选项，插件开发者就可以汇总一系列构建的输出；例如，创建一个插件用来汇总一次构建包含的所有项目的报告。aggregator设置成true的目标只针对Maven构建的顶层项目运行。该注解默认的值是false。

@requiresOnline (true|false)
当该注解的值是true的时候，Maven在脱机模式运行的时候该目标运行就会失败。Maven会抛出一个错误。默认值：false。

@requiresDirectInvocation
当设置成true的时候，只有当用户显式的从命令行触发的时候，该插件才能得以执行。如果有人试图将其绑定到一个生命周期阶段，Maven就会抛出一个错误。默认值是false。

@phase <phaseName>
该注解指定目标默认的生命周期阶段。如果你将该目标的执行配置到了pom.xml而且没有指定一个阶段。Maven就会使用该注解的值将其绑定到一个默认的阶段。

@execute [goal=goalName|phase=phaseName [lifecycle=lifecycleId]]
该注解有很多种使用方式。如果提供了一个阶段，Maven就会执行一个平行的生命周期（直到指定的阶段）。这个单独执行的结果可以通过Maven属性${executedProperty}供插件使用。

第二种使用该注解的方式是使用prefix:goal标记指定一个显式的目标。当你仅仅指定一个目标的时候，Maven会在一个平行的环境中执行该目标，不会影响当前的Maven构建。

第三种使用该注解的方式是，使用一个生命周期定义文件，并指定这个生命周期的一个阶段。

```
@execute phase="package" lifecycle="zip"
@execute phase="compile"
@execute goal="zip:zip"
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

如果你看一下EchoMojo的源码，你会注意到Maven并没有使用Java 5的标准注解。而是使用了Commons Attributes[6]。在注解成为Java语言的一部分之前，Commons Attributes为Java程序员提供了一种使用注解的方式。为什么Maven不使用Java 5的注解呢？这是因为Maven是针对Java 5之前的JVM设计的。因为Maven必须支持Java比较老的版本，所以它不能使用任何Java 5的新特性。

17.4.6. 当Mojo失败的时候

Mojo中的execute()方法抛出两个异常，MojoExecutionException和MojoFailureException。这两个异常的区别既微妙又重要，这要看当目标运行"失败"的时候发生了什么。一个MojoExecutionException应该是一个致命的异常，发生了一些不可恢复的错误。如果有什么事情导致构建完全终止，你就需要抛出一个MojoExecutionException；比如说你正试图往磁盘写数据，但没有可用空间，或者，你试图发布构件到一个远程仓库，但是连接不了远程服务器。如果没有机会继续构建，就抛出一个MojoExecutionException；发生了一些严重的事情，你希望停止构建并让用户看到"BUILD ERROR"信息。

而MojoFailureException就相对没有那么严重，一个目标可以失败，但它可能并不是Maven构建的世界末日。一个单元测试可以失败，或者MD5校验和可以失败；两者都是潜在的问题，但是你不会想要抛出一个异常去终止整个构建。在这种情况下，你可以抛出一个MojoFailureException。当Maven遇到项目失败的时候，他会提供不同的"弹性"设置。如下所述：

当你运行一个Maven构建的时候，它会包含一系列的项目，每个项目可以成功或者失败。你可以三种可选的失败模式：

mvn -ff
: 最快失败模式：Maven会在遇到第一个失败的时候失败（停止）。

mvn -fae
: 最后失败模式：Maven会在构建最后失败（停止）。如果Maven refactor中一个失败了，Maven会继续构建其它项目，并在构建最后报告失败。

mvn -fn
: 从不失败模式：Maven从来不会为一个失败停止，也不会报告失败。

如果你正在运行一个持续集成构建，无论单个项目构建成败与否都要继续构建，你可能想要忽略失败。作为一个插件开发者，你必须根据你某个特定的条件来判断使用MojoExecutionException还是MojoFailureExeception。

[6] http://commons.apache.org/attributes/

17.5. Mojo参数

通过参数配置Mojo，和execute()方法及Mojo注解相比同样重要。本节讲述有关Mojo参数的配置和主题。

17.5.1. 为Mojo参数提供值

在EchoMojo中我们使用如下的注解声明了message参数。

```
/**
 * Any Object to print out.
 * @parameter
 *       expression="${echo.message}"
 *       default-value="Hello Maven World"
 */
private Object message;
```

该参数的默认表达式是${echo.message}，意思是Maven会使用echo.message属性的值来设置message的值。如果echo.message属性的值是null，@parameter注解的default-value属性就会被使用。除了使用echo.message属性，我们也可以在项目的POM中配置EchoMojo的message参数的值。

有很多中方式可以填充EchoMojo的message参数的值。首先我们可以从命令行传入一个值（假设你已经将org.sonatype.mavenbook.plugins添加到你的pluginGroups中）：

```
$ mvn first:echo -Decho.message="Hello Everybody"
```

我们也可以通过在POM或者settings.xml中设定一个属性来指定该message参数的值：

```
<project>
  ...
  <properties>
    <echo.message>Hello Everybody</echo.message>
  </properties>
</project>
```

该参数还可以直接通过配置插件来进行配置。如果我们想要直接自定义message参数，我们可以使用如下的build配置。下面的配置绕开了echo.message属性，而是在插件配置中填充Mojo参数：

```
<project>
  ...
  <build>
    <plugins>
      <plugin>
        <groupId>org.sonatype.mavenbook.plugins</groupId>
        <artifactId>first-maven-plugin</artifactId>
```

```
        <version>1.0-SNAPSHOT</version>
        <configuration>
          <message>Hello Everybody!</message>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

如果我们想要在一个生命周期的不同阶段中运行EchoMojo两次，并且希望每次运行都能自定义message参数，我们可以在如下在POM中的execution级别配置这个参数值：

```
<build>
  <build>
    <plugins>
      <plugin>
        <groupId>org.sonatype.mavenbook.plugins</groupId>
        <artifactId>first-maven-plugin</artifactId>
        <version>1.0-SNAPSHOT</version>
        <executions>
          <execution>
            <id>first-execution</id>
            <phase>generate-resources</phase>
            <goals>
              <goal>echo</goal>
            </goals>
            <configuration>
              <message>The Eagle has Landed!</message>
            </configuration>
          </execution>
          <execution>
            <id>second-execution</id>
            <phase>validate</phase>
            <goals>
              <goal>echo</goal>
            </goals>
            <configuration>
              <message>0.6-SNAPSHOT</message>
            </configuration>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</build>
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

虽然上面的配置样例看起来有些啰嗦，但它展示了Maven的弹性。在前面的配置样例中，我们将EchoMojo同时绑定到了默认Maven生命周期的validate和generate-resources阶段。第一次执行被绑定到了generate-resources，它为message参数提供了字符串值“The Eagle has Landed!”。第二次运行则被绑定到了validate阶段，它提供了一个属性引用0.6-SNAPSHOT。当你为该项目运行mvn install的时候，你会看到first:echo目标执行了两次，并打印了不同的信息。

17.5.2. 多值的Mojo参数

插件的参数可以接受多于一个的值。看一下例 17.7 “一个带有多值参数的插件”中的ZipMojo。参数includes和excludes都是多值的String数组，它们为一个创建ZIP文件的组件指定了包含和排除模式。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

```
/**
 * Zips up the output directory.
 * @goal zip
 * @phase package
 */
public class ZipMojo extends AbstractMojo
{
    /**
     * The Zip archiver.
     * @parameter expression="${component.org.codehaus.plexus.archiver.Archiver#zip
     */
    private ZipArchiver zipArchiver;

    /**
     * Directory containing the build files.
     * @parameter expression="/usr/local/hudson/hudson-home/jobs/maven-guide-zh-to-
     */
    private File buildDirectory;

    /**
     * Base directory of the project.
     * @parameter expression="/usr/local/hudson/hudson-home/jobs/maven-guide-zh-to-
     */
    private File baseDirectory;

    /**
     * A set of file patterns to include in the zip.
     * @parameter alias="includes"
     */
    private String[] mIncludes;

    /**
     * A set of file patterns to exclude from the zip.
     * @parameter alias="excludes"
     */
    private String[] mExcludes;

    public void setExcludes( String[] excludes ) { mExcludes = excludes; }

    public void setIncludes( String[] includes ) { mIncludes = includes; }

    public void execute()
        throws MojoExecutionException
    {
        try {
            zipArchiver.addDirectory( buildDirectory, includes, excludes );
            zipArchiver.setDestFile( new File( baseDirectory, "output.zip" ) );
            zipArchiver.createArchive();
        } catch( Exception e ) {
            throw new MojoExecutionException( "Could not zip", e );
        }
    }
}
```

要配置一个多值的Mojo参数，你应该为这类参数使用一组元素。如果这个多值参数的名称是includes，你就可以使用一个includes元素，它包含一组include子元素。如果这个多值参数的名称是excludes，你就应该使用带有exclude子元素的excludes元素。要配置ZipMojo使其忽略所有以.txt及波浪号结尾的文件，你可以使用如下的插件配置。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛


```
<project>
  ...
  <build>
    <plugins>
      <plugin>
        <groupId>org.sonatype.mavenbook.plugins</groupId>
        <artifactId>zip-maven-plugin</artifactId>
        <configuration>
          <excludes>
            <exclude>**/*.txt</exclude>
            <exclude>**/*~</exclude>
          </excludes>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

17.5.3. 依赖于一个Plexus组件

一个Mojo是一个由名为Plexus的Ioc容器管理的组件。Mojo可以通过使用`@parameter` `#@component`注解声明Mojo参数，然后依赖于其它Plexu管理的组件。例 17.7 “一个带有多值参数的插件”展示了`ZipMojo`使用`@parameter`注解依赖于一个Plexus组件，该依赖也可以使用`@component`注解来声明。

例 17.8. 依赖于一个Plexus组件

```
/**
 * The Zip archiver.
 * @component role="org.codehaus.plexus.archiver.Archiver" roleHint="zip"
 */
private ZipArchiver zipArchiver;
```

当Maven初始化该Mojo的时候，它会尝试通过指定的role和role hint来获取Plexus组件。在该例中，这个Mojo关联到一个ZipArchiver组件，后者能帮助`ZipMojo`创建ZIP文件。

17.5.4. Mojo参数注解

除非你坚持要手工编写自己的插件描述符，否则你完全不需要编写那个XML文件。Maven Plugin插件有一个`plugin:descriptor`目标绑定到了generate-resources阶段。该目标根据Mojo的注解生成插件描述符。要配置Mojo参数，你应该使用下列的注解，将其声明到

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

私有成员变量上。你也可以在公有setter方法上使用这些注解，但Maven插件一般的约定是直接注解私有成员变量。

@parameter [alias="someAlias"] [expression="${someExpression}"] [default-value="value"]
: 标记一个私有字段（或者一个setter方法）为一个参数。alias提供该参数的名称。如果没有alias，Maven会使用变量名为参数名。expression是一个Maven用来计算并获值的一个表达式。通常这个表达式只是一个属性引用如${echo.message}。default-value是当表达式不能求得值，POM的插件配置中也没有显式提供时，Mojo会使用的值。

@required
: 如果使用了该注解，那么在该Mojo运行前该参数就必须要有一个可用的值。如果Maven试图运行该Mojo的时候该参数的值为null，Maven就会抛出一个错误。

@readonly
: 如果使用该注解，用户就不能从POM直接配置这个参数。你就需要parameter注解的expression属性。例如，如果你想要确保一个特定参数的值永远是POM中finalName属性的值，你就可以使用表达式book，并添加这个@readOnly注解。这样，用户就只能通过更改POM中finalName的值来更改这个参数的值。

@component
: 告诉Maven使用Plexus组件填充该字段。一个正确的@component注解值如下：

```
@component role="org.codehaus.plexus.archiver.Archiver" roleHint="zip"
```

该配置的效果是可以从Plexus获得ZipArchiver组件。ZipArchiver是一个对应role hint为zip的Archiver组件。除了@component注解，你也可以使用@parameter注解和一个expression属性，如：

```
@parameter expression="${component.org.codehaus.plexus.archiver.Archiver#zip}
```

虽然两种注解的效果一样，但对于配置Plexus组件依赖来说，@component注解是更推荐的方式。

@deprecated
: 该参数以过期，不再推荐使用。用户可以继续配置该参数，但会得到一条警告信息。

17.6. 插件和Maven生命周期

在第 10 章构建生命周期中，你学习了生命周期可以通过打包类型定制。一个插件可以同时引入一个新的打包类型，并自定义该生命周期。在本节中，我们将会学习如果通

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

过一个自定义的Maven插件来自定义生命周期。你同时也会看到如何让Mojo在平行的生命周期中执行。

17.6.1. 执行平行的生命周期

让我们假设你编写了一些目标，它们依赖于前一个构建的输出。也许`ZipMojo`目标只有在拥有归档输出的前提下才能运行。你可以使用Mojo类的`@execute`注解来指定这样的前置目标。该注解会让Maven生成一个平行的构建，并在这个平行的Maven实例中执行一个目标或者生命周期，该运行并不会影响当前构建。也许你编写了一个Mojo，它每天只运行一次，它首先会运行mvn install，然后将所有的输出打包至某种自定义的分发格式。你的Mojo描述符可以告诉Maven，在运行这个自定义Mojo之前，你想要运行默认生命周期的所有直到install的阶段，然后以属性`${executedProject}`的形式将项目结果暴露给你的Mojo。然后你就可以在进行某种后期处理之前引用那个项目的属性。

另一种可能性是，你有一个目标做一些与默认生命周期完全无关的工作。让我们考虑这样一个例子，你有一个目标使用LAME将WAV文件转换成MP3，但在这之前你需要运行一个生命周期，将MIDI文件转换成WAV。（你可以用Maven做任何事，这里的例子并没有太“离谱”）你已经创建了一个“midi-sound”生命周期，现在你想要包含midi-sound生命周期install阶段的输出，而现在你自己的web应用项目的打包类型是war。由于你的项目在war打包类型生命周期运行，你需要让mojo能够fork一个完全独立的，使用midi-sound的生命周期。你可以通过在你的mojo上添加注解`@execute lifecycle="midi-source" phase="install"`来达到这样的目的。

@execute goal="<goal>"
: 会在执行当前目标之前运行声明的目标。目标名称使用`prefix:goal`标记指定。

@execute phase="<phase>"
: 在执行当前生命周期之前，fork出另一个构建生命周期（直到指定的阶段）。如果没有指定生命周期，Maven会使用当前构建的生命周期。

@execute lifecycle="<lifecycle>" phase="<phase>"
: 会执行给定的生命周期。自定义的生命周期可以在`META-INF/maven/lifecycle.xml`中定义。

17.6.2. 创建自定义的生命周期

自定义生命周期必须在插件项目的`META-INF/maven/lifecycle.xml`文件中定义。你可以引入这个定义在`src/main/resources`下的`META-INF/maven/lifecycle.xml`文件。以下的`lifecycle.xml`声明了一个名为zipcycle的生命周期，它在package阶段包含了一个zip目标。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

例 17.9. 在lifecycle.xml中自定义生命周期

```
<lifecycles>
  <lifecycle>
    <id>zipcycle</id>
    <phases>
      <phase>
        <id>package</id>
        <executions>
          <execution>
            <goals>
              <goal>zip</goal>
            </goals>
          </execution>
        </executions>
      </phase>
    </phases>
  </lifecycle>
</lifecycles>
```

如果你想要在另一个构建中运行这个zipcycle生命周期，你可以创建一个ZipForkMojo，然后使用@execute注解来告诉Maven，当ZipForkMojo运行的时候，逐步通过zipcycle生命周期的阶段。

例 17.10. 在Mojo中Fork一个自定义生命周期

```
/**
 * Forks a zip lifecycle.
 * @goal zip-fork
 * @execute lifecycle="zipcycle" phase="package"
 */
public class ZipForkMojo extends AbstractMojo
{
  public void execute()
    throws MojoExecutionException
  {
    getLog().info( "doing nothing here" );
  }
}
```

运行ZipForkMojo的时候会fork出另一个生命周期。如果你配置了你插件的前缀为zip，运行zip-fork会得到类似于如下输出的结果：

```
$ mvn zip:zip-fork
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛


```
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'zip'.
[INFO] ----------------------------------------------------------------------------
[INFO] Building Maven Zip Forked Lifecycle Test
[INFO]    task-segment: [zip:zip-fork]
[INFO] ----------------------------------------------------------------------------
[INFO] Preparing zip:zip-fork
[INFO] [site:attach-descriptor]
[INFO] [zip:zip]
[INFO] Building zip: ~/maven-zip-plugin/src/projects/zip-lifecycle-test/target/outp
[INFO] [zip:zip-fork]
[INFO] doing nothing here
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 1 second
[INFO] Finished at: Sun Apr 29 16:10:06 CDT 2007
[INFO] Final Memory: 3M/7M
[INFO] ------------------------------------------------------------------------
```

调用zip-fork会衍生出另外一个生命周期，Maven运行zipcycle生命周期，然后从ZipFormMojo的execute方法打印信息。

17.6.3. 覆盖默认生命周期

你已经创建了自己的生命周期，并能在一个Mojo中将其衍生使用。下一个你可能要问的问题是，如何覆盖默认的生命周期。如何如何创建定制的生命周期并将其绑定到项目上？在第 10 章构建生命周期中，我们看到了项目的打包类型定义了项目的生命周期。所有打包类型都有差异；war绑定不同的目标到package阶段，自定义的生命周期如来自Israfil Flex 3的swf绑定不同的目标到compile阶段。在你创建自定义生命周期的时候，你可以通过为插件提供一些Plexus配置，将生命周期绑定到特定的打包类型。

要为新的生命周期定义新的打包类型，你需要配置Plexus中的LifecycleMapping组件。在你的插件项目中，在src/main/resources下创建一个META-INF/plexus/components.xml文件。在这个components.xml中添加如例 17.11 “覆盖默认生命周期”的内容。在role-hint下设置打包类型的名称，在pheases中设置包含所有需要绑定目标的坐标（省略version）。多个目标可以用逗号分隔，绑定到同一个阶段。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

例 17.11. 覆盖默认生命周期

```
<component-set>
  <components>
    <component>
      <role>org.apache.maven.lifecycle.mapping.LifecycleMapping</role>
      <role-hint>zip</role-hint>
      <implementation>org.apache.maven.lifecycle.mapping.DefaultLifecycleMapping</i
      <configuration>
        <phases>
          <process-resources>org.apache.maven.plugins:maven-resources-plugin:resour
          <compile>org.apache.maven.plugins:maven-compiler-plugin:compile</compile>
          <package>org.sonatype.mavenbook.plugins:maven-zip-plugin:zip</package>
        </phases>
      </configuration>
    </component>
  </components>
</component-set>
```

如果你创建了一个插件，它定义了新的打包类型和定制的生命周期，Maven在将你的插件添加到项目POM中并设置extensions元素为true之前，对你插件的定义一概不知。只有在那之后，Maven才会扫描你的插件，而不仅仅只是运行Mojo，它会查找`META-INF/plexus`下的`components.xml`文件，然后它会在你的项目中使新的打包类型可用。

例 17.12. 作为一个Extension配置一个插件

```
<project>
  ...
  <build>
    ...
    <plugins>
      <plugin>
        <groupId>com.training.plugins</groupId>
        <artifactId>maven-zip-plugin</artifactId>
        <extensions>true</extensions>
      </plugin>
    </plugins>
  </build>
</project>
```

一旦你添加了插件，并将extensions元素设置成true，你就可以使用这个定制的打包类型，你的项目就能运行关联到该打包类型的定制生命周期。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

第 18 章 使用可选语言编写插件

你可以使用Java编写Mojo，你也可以使用其它可选的语言。Maven支持了很多语言来编写插件，本章介绍如何使用三种语言：Groovy, Ant, 及Ruby，来编写插件。

18.1. 使用Ant编写插件

Ant从某种意义上来说不是一种语言，它是一种构建工具，能让你通过一组归类成构建目标的任务来描述一个构建。Ant允许你声明不同构建目标之间的依赖；比如，使用Ant你实际上是在创建自己的生命周期。一个Ant的build.xml可能会有一个install目标，它依赖于一个test目标，而后者又依赖于compile目标。Ant可以说是Maven的前辈，它是一个普遍的过程式构建工具，在Maven引入高度可重用的构建插件及全局生命周期之前，Ant几乎被所有项目使用。

虽然Maven比Ant进步了，但是在描述构建过程的某些部分的时候，Ant仍然十分有用。当你需要执行一些文件操作，XSLT转换，或者其它你能可能想到的操作的时候，Ant提供的一组任务便十分简洁有效。关于Ant任务有一个很大的类库，包括了几乎所有功能，运行JUnit测试，转换XML，使用SCP复制文件至远程服务器，等等。关于所有可用的Ant任务列表可以从这里查到：Apache Ant 手册[1]。你可以使用这些任务作为一个低级别的构建自定义语言，你也可以不用Java而是使用Ant编写一个Maven插件，你可以给一个Ant构建目标的Mojo传入参数。

18.2. 创建一个Ant插件

要使用Ant创建一个Maven插件，你需要有一个pom.xml和一个使用Ant实现的Mojo。一开始，创建一个名为firstant-maven-plugin的项目目录。在该目录创建如下的pom.xml。

[1] http://ant.apache.org/manual/tasksoverview.html

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

例 18.1. 一个Ant Maven插件的POM

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.sonatype.mavenbook.plugins</groupId>
  <artifactId>firstant-maven-plugin</artifactId>
  <name>Example Ant Mojo - firstant-maven-plugin</name>
  <packaging>maven-plugin</packaging>
  <version>1.0-SNAPSHOT</version>
  <dependencies>
    <dependency>
      <groupId>org.apache.maven</groupId>
      <artifactId>maven-script-ant</artifactId>
      <version>2.0.6</version>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-plugin-plugin</artifactId>
        <version>2.3</version>
        <dependencies>
          <dependency>
            <groupId>org.apache.maven</groupId>
            <artifactId>maven-plugin-tools-ant</artifactId>
            <version>2.0.5</version>
          </dependency>
        </dependencies>
      </plugin>
    </plugins>
  </build>
</project>
```

紧接着，你需要创建你的Ant Mojo。一个Ant Mojo包括两个部分：XML文件中的Ant任务，以及一个提供Mojo描述符信息的文件。Ant插件工具会到`/usr/local/hudson/hudson-home/jobs/maven-guide-zh-to-production/workspace/content-zh/src/main/scripts`目录下寻找这些文件。在这里创建一个文件名为echo.build.xml，包含如下的Ant XML。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

例 18.2. Echo Ant Mojo

```
<project>
  <target name="echotarget">
    <echo>${message}</echo>
  </target>
</project>
```

另外创建一个文件描述这个Echo Ant Mojo，同样的在/usr/local/hudson/hudson-home/jobs/maven-guide-zh-to-production/workspace/content-zh/src/main/scripts目录下，名为echo.mojos.xml。

例 18.3. Echo Ant Mojo描述符

```
<pluginMetadata>
  <mojos>
    <mojo>
      <goal>echo</goal>
      <call>echotarget</call>
      <description>Echos a Message</description>
      <parameters>
        <parameter>
          <name>message</name>
          <property>message</property>
          <required>false</required>
          <expression>${message}</expression>
          <type>java.lang.Object</type>
          <defaultValue>Hello Maven World</defaultValue>
          <description>Prints a message</description>
        </parameter>
      </parameters>
    </mojo>
  </mojos>
</pluginMetadata>
```

这个echo.mojos.xml文件为该插件配置Mojo描述符。它提供了一个名为"echo"的目标名称，它在call元素中告诉Maven调用哪个Ant任务。除了配置描述信息，该XML还使用表达式${message}配置了message参数，并且为此提供了一个缺省值"Hello Maven World"。

如果你在~/.m2/settings.xml中配置你的插件组包含了org.sonatype.mavenbook.plugins，你就可以通过执行如下的命令安装这个Ant插件：

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛


```
$ mvn install
[INFO] ------------------------------------------------------------------------
[INFO] Building Example Ant Mojo - firstant-maven-plugin
[INFO]    task-segment: [install]
[INFO] ------------------------------------------------------------------------
[INFO] [plugin:descriptor]
[INFO] Using 3 extractors.
[INFO] Applying extractor for language: java
[INFO] Extractor for language: java found 0 mojo descriptors.
[INFO] Applying extractor for language: bsh
[INFO] Extractor for language: bsh found 0 mojo descriptors.
[INFO] Applying extractor for language: ant
[INFO] Extractor for language: ant found 1 mojo descriptors.
...
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
```

注意plugin:descriptor目标发现了一个Ant Mojo描述符。要运行这个目标，你可以执行如下的命令：

```
$ mvn firstant:echo
...
[INFO] [firstant:echo]

echotarget:
     [echo] Hello Maven World
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
```

echo目标执行后打印了默认的message参数的值。如果你习惯了Apache Ant构建脚本，你会注意到Ant打印了所执行目标的名称，并且在echo任务的输出上加了一个日志前缀。

18.3. 使用JRuby编写插件

Ruby是一种面向对象脚本语言，它为元编程（meta-programming）和反射提供了丰富的基础设施。Ruby对于closure和block的使用造就了一种既紧凑又不失强大的编程风格。虽然Ruby自1993年就出现了，但大部分人了解到Ruby是在基于Ruby的web框架，Ruby on Rails流行之后。JRuby是用Java编写的Ruby解析器。要了解更多的关于Ruby语言的信息，可参考http://www.ruby-lang.org/，而关于JRuby，可参考http://jruby.codehaus.org/。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

18.3.1. 创建一个JRuby插件

要使用JRuby创建一个Maven插件，你需要一个pom.xml和一个Ruby实现的Mojo。一开始，创建一个名为firstruby-maven-plugin的项目目录，在该目录中创建如下的pom.xml。

例 18.4. 一个JRuby Maven插件的POM

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.sonatype.mavenbook.plugins</groupId>
  <artifactId>firstruby-maven-plugin</artifactId>
  <name>Example Ruby Mojo - firstruby-maven-plugin</name>
  <packaging>maven-plugin</packaging>
  <version>1.0-SNAPSHOT</version>
  <dependencies>
    <dependency>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>jruby-maven-plugin</artifactId>
      <version>1.0-beta-4</version>
      <scope>runtime</scope>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-plugin-plugin</artifactId>
        <version>2.3</version>
        <dependencies>
          <dependency>
            <groupId>org.codehaus.mojo</groupId>
            <artifactId>jruby-maven-plugin</artifactId>
            <version>1.0-beta-4</version>
          </dependency>
        </dependencies>
      </plugin>
    </plugins>
  </build>
</project>
```

紧接着，你需要创建一个由Ruby实现的Mojo。Maven会到/usr/local/hudson/hudson-home/jobs/maven-guide-zh-to-production/workspace/content-zh/src/main/scripts目录寻找Ruby Mojo。创建如下的/usr/local/hudson/hudson-home/jobs/maven-guide-zh-to-production/workspace/content-zh/src/main/scripts/echo.rb文件：

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

例 18.5. Echo Ruby Mojo

```
# Prints a message
# @goal "echo"
# @phase "validate"
class Echo < Mojo

  # @parameter type="java.lang.String" default-value="Hello Maven World" expression
  def message
  end

  def execute
    info $message
  end

end

run_mojo Echo
```

这个Echo类必须扩展Mojo类，它必须重写execute()方法。在echo.rb文件最后，你将需要使用"run_mojo Echo"运行这个mojo。要安装该插件，运行mvn install：

```
$ mvn install
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------------------
[INFO] Building Example Ruby Mojo - firstruby-maven-plugin
[INFO]    task-segment: [install]
[INFO] ------------------------------------------------------------------------
...
[INFO] [plugin:descriptor]
...
[INFO] Applying extractor for language: jruby
[INFO] Ruby Mojo File: /echo.rb
[INFO] Extractor for language: jruby found 1 mojo descriptors.
...
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
```

在构建过程中，你会看到Maven Plugin插件的descriptor目标应用了JRuby萃取器(extractor)创建了一个plugin.xml文件，该文件萃取了所有Echo类的注解。如果你已经配置你默认的插件组包含了org.sonatype.plugins，你就可以使用如下的命令运行该目标：

```
$ mvn firstruby:echo
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛


```
...
[INFO] [firstruby:echo]
[INFO] Hello Maven World
...
```

18.3.2. Ruby Mojo实现

Ruby Mojo在源文件中使用注释进行注解。一个单独的注解如@parameter可以接受很多属性，而多个属性都必须在同一行中指定。注解属性之间不能有换行符。类和参数都可以被注解。关于参数可以使用四种注解：@parameter，@required，@readonly，和@deprecated。@parameter注解接受如下的属性：

alias
: 参数的别名。可以同样用来填充该参数的可选的名称。

default-value
: 如果参数被提供的值或者参数表达式的结果是null，就使用该缺省值。在echo.rb中，我们指定了缺省值为"Hello Maven World"。

expression
: 指定一个表达式，可以用来解析Maven属性或者系统属性。

type
: 该参数的完全限定的Java类型。如果没有指定，默认值为java.lang.String。

除了@parameter注解，参数还可以使用如下的注解：

@required "<true|false>"
: 标记一个参数是否为必须的。默认值是false。

@readonly "<true|false>"
: 标记参数为只读。如果为true，你就无法从命令行覆盖默认值或者默认表达式的值。这里，默认值为false。

@deprecated "<true|false>"
: 标记该参数是废弃的。默认为false。

将所有这些内容放在一起，一个echo.rb中完全注解的message参数看起来就会成为这个样子：

```
# @parameter type="java.lang.String" default-value="Hello Maven World" expression="
# @readonly true
# @required false
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛


```
# @deprecated false
def message
end
```

Ruby的Mojo类可以使用如下的注解：

@goal
: 指定目标的名称。

@phase
: 该目标绑定的默认的阶段。

@requiresDependencyResolution
: 如果Mojo需要依赖在运行之前被解析，则为true。

@aggregator
: 标记该mojo为聚合。

@execute
: 支持在运行Mojo之前执行一个目标或者生命周期阶段。@execute注解接受如下的属性；

  goal
  : 要执行的目标的名称。

  phase
  : 要执行的生命周期阶段的名称。

  lifecycle
  : 生命周期的名称（如果不是默认的话）。

举一个注解Mojo类的例子，代码如下：

```
# Completes some build task
# @goal custom-goal
# @phase install
# @requiresDependencyResolution false
# @execute phase=compile
class CustomMojo < Mojo
  ...
end
```

Mojo参数可以引用Java类和Maven属性。下面的例子展示了如何从Ruby Mojo访问MavenProject对象。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

例 18.6. 从Ruby Mojo中引用MavenProject对象

```
# This is a mojo description
# @goal test
# @phase validate
class Test < Mojo
  # @parameter type="java.lang.String" default-value="nothing" alias="a_string"
  def prop
  end

  # @parameter type="org.apache.maven.project.MavenProject" expression="${project}"
  # @required true
  def project
  end

  def execute
    info "The following String was passed to prop: '#{$prop}'"
    info "My project artifact is: #{$project.artifactId}"
  end
end

run_mojo Test
```

在前面的例子中，我们可以使用标准的Ruby语法访问Project类的属性。如果你将test.rb放到firstruby-maven-plugin的`src/main/scripts`目录，然后install这个插件，接着运行它，你会看到如下的输出：

```
$ mvn install
...
[INFO] [plugin:descriptor]
[INFO] Using 3 extractors.
[INFO] Applying extractor for language: java
...
[INFO] Applying extractor for language: jruby
[INFO] Ruby Mojo File: /echo.rb
[INFO] Ruby Mojo File: /test.rb
[INFO] Extractor for language: jruby found 2 mojo descriptors.
...
$ mvn firstruby:test
...
[INFO] [firstruby:test]
[INFO] The following String was passed to prop: 'nothing'
[INFO] My project artifact is: firstruby-maven-plugin
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

18.3.3. Ruby Mojo中使用日志

要在Ruby Mojo中使用日志，调用info()，debug()，和error()方法，并传入日志信息参数。

```
# Tests Logging
# @goal logtest
# @phase validate
class LogTest < Mojo

  def execute
    info "Prints an INFO message"
    error "Prints an ERROR message"
    debug "Prints to the Console"
  end

end

run_mojo LogTest
```

18.3.4. Raise一个MojoError

如果在一个Ruby Mojo中有一个未发现的错误，你就需要raise一个MojoError。例 18.7 “在Ruby Mojo中raise一个MojoError”展示了如何raise一个MojoError。例子中的mojo首先打印了一条日志信息，然后raise一个MojoError。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

例 18.7. 在Ruby Mojo中raise一个MojoError

```
# Prints a Message
# @goal error
# @phase validate
class Error < Mojo

  # @parameter type="java.lang.String" default-value="Hello Maven World" expression=
  # @required true
  # @readonly false
  # @deprecated false
  def message
  end

  def execute
    info $message
    raise MojoError.new( "This Mojo Raised a MojoError" )
  end

end

run_mojo Error
```

运行该Mojo，会产生如下的输出：

```
$ mvn firstruby:error
...
INFO] [firstruby:error]
[INFO] Hello Maven World
[ERROR] This Mojo Raised a MojoError
```

18.3.5. 在JRuby中引用Plexus组件

Ruby Mojo可以依赖于一个Plexus组件。为此，你需要使用@parameter注解的expression参数，在这里为Plexus指定role和hint。下面的例子中，Ruby Mojo依赖于Archiver组件，Maven会据此从Plexus获取该组件。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

例 18.8. 在Ruby Mojo中依赖于一个Plexus组件

```
# This mojo tests plexus integration
# @goal testplexus
# @phase validate
class TestPlexus < Mojo

  # @parameter type="org.codehaus.plexus.archiver.Archiver" \
expression="${component.org.codehaus.plexus.archiver.Archiver#zip}"
  def archiver
  end

  def execute
    info $archiver
  end
end

run_mojo TestPlexus
```

请注意Ruby Mojo注解的属性不能跨越多行。如果你要运行该目标，你会看到Maven将根据role：org.codehaus.plexus.archiver.Archiver和hit：zip尝试从Plexus获取一个组件。

18.4. 使用Groovy编写插件

Groovy是一个基于Java虚拟机的动态语言，它最终也被编译成Java字节码。Groovy是Codehaus社区的项目。如果你十分熟悉Java，Groovy就会是一种很自然的脚本语言。Groovy使用了Java的很多特性，同时进行了略微的裁剪，并加入了一些新特性，如闭包(closure)，鸭子类型判断(duck-typing)，和正则表达式。要了解更多的关于Groovy的信息，可以访问Groovy的站点：http://groovy.codehaus.org。

18.4.1. 创建一个Groovy插件

要使用Groovy创建一个Maven插件，你只需要两个文件：一个pom.xml和一个用Groovy实现的Mojo。首先，创建一个名为firstgroovy-maven-plugin的项目目录，然后在该目录下创建如下的pom.xml文件：

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

例 18.9. Groovy Maven插件的POM

```
<?xml version="1.0" encoding="UTF-8"?>
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.sonatype.mavenbook.plugins</groupId>
  <artifactId>firstgroovy-maven-plugin</artifactId>
  <name>Example Groovy Mojo - firstgroovy-maven-plugin</name>
  <packaging>maven-plugin</packaging>
  <version>1.0-SNAPSHOT</version>
  <dependencies>
    <dependency>
      <groupId>org.codehaus.mojo.groovy</groupId>
      <artifactId>groovy-mojo-support</artifactId>
      <version>1.0-beta-3</version>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-plugin-plugin</artifactId>
        <version>2.3</version>
      </plugin>
      <plugin>
        <groupId>org.codehaus.mojo.groovy</groupId>
        <artifactId>groovy-maven-plugin</artifactId>
        <version>1.0-beta-3</version>
        <extensions>true</extensions>
        <executions>
          <execution>
            <goals>
              <goal>generateStubs</goal>
              <goal>compile</goal>
              <goal>generateTestStubs</goal>
              <goal>testCompile</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

这个POM到底是怎么回事呢？首先，注意这个POM的打包类型是maven-plugin，这是因为我们要创建一个打包为Maven插件的项目。接着，注意该项目依赖于groupId为org.codehaus.mojo.groovy，artifactId为groovy-maven-plugin的构件。

接着在项目的`src/main/groovy`目录，创建一个名为EchoMojo.groovy的文件，其包含如下的EchoMojo类：

例 18.10. Echo Groovy Mojo

```
package org.sonatype.mavenbook.plugins

import org.codehaus.mojo.groovy.GroovyMojo

/**
 * Example goal which echos a message
 *
 * @goal echo
 */
class EchoMojo extends GroovyMojo {

    /**
     * Message to print
     *
     * @parameter expression="${echo.message}"
     *            default-value="Hello Maven World"
     */
    String message

    void execute() {
      log.info( message )
    }
}
```

附录 A. 附录：Settings细节

A.1. 简介

settings.xml文件中的settings元素包含了很多子元素，它们定义的值被用来配置Maven的执行情况。该settings文件的设置会被应用到很多个项目上，因此这里的设置不应该和任何一个特定的项目绑定，并且该设置的内容也不应该分发给它人。该文件定义的值包括本地仓库地址，候选的远程仓库仓库服务器，以及一些认证信息。settings.xml文件可位于两个地方：

- Maven安装目录：$M2_HOME/conf/settings.xml
- 用户特定的Settings文件：~/.m2/settings.xml

这里是settings元素下最顶层元素的概览：

例 A.1. settings.xml中顶层元素的概览

```
<settings xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                      http://maven.apache.org/xsd/settings-1.0.0.xsd">
  <localRepository/>
  <interactiveMode/>
  <usePluginRegistry/>
  <offline/>
  <pluginGroups/>
  <servers/>
  <mirrors/>
  <proxies/>
  <profiles/>
  <activeProfiles/>
</settings>
```

A.2. Settings细节

A.2.1. 简单值

一半顶层settings元素是简单值，它们表示的一系列值可以配置Maven的核心行为：

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

例 A.2. settings.xml中的简单顶层元素

```
<settings xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                      http://maven.apache.org/xsd/settings-1.0.0.xsd">
  <localRepository>/ebs1/build-machine/usr/local/hudson/hudson-home/jobs/maven-guid
  <interactiveMode>true</interactiveMode>
  <usePluginRegistry>false</usePluginRegistry>
  <offline>false</offline>
  <pluginGroups>
    <pluginGroup>org.codehaus.mojo</pluginGroup>
  </pluginGroups>
  ...
</settings>
```

这些简单顶层元素是:

localRepository
: 该值表示构建系统本地仓库的路径。其默认值为`/ebs1/build-machine/usr/local/hudson/hudson-home/jobs/maven-guide-zh-to-production/workspace/.m2/repository`。

interactiveMode
: 如果Maven需要和用户交互以获得输入，则设置成`true`，反之则应为`false`。默认为`true`。

usePluginRegistry
: 如果需要让Maven使用文件`/ebs1/build-machine/usr/local/hudson/hudson-home/jobs/maven-guide-zh-to-production/workspace/.m2/plugin-registry.xml`来管理插件版本，则设为`true`。默认为`false`。

offline
: 如果构建系统需要在离线模式下运行，则为`true`，默认为`false`。当由于网络设置原因或者安全因素，构建服务器不能连接远程仓库的时候，该配置就十分有用。

pluginGroups
: 该元素包含一个`pluginGroup`元素列表，每个子元素包含了一个`groupId`。当我们使用某个插件，并且没有在命令行为其提供`groupId`的时候，Maven就会使用该列表。默认情况下该列表包含了`org.apache.maven.plugins`。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

A.2.2. 服务器（Servers）

POM中的distributionManagement元素定义了部署的仓库。然而，一些设置如安全证书不应该和pom.xml一起分发。这种类型的信息应该存在于构建服务器上的settings.xml文件中。

例 A.3. settings.xml中的Server配置

```
<settings xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                      http://maven.apache.org/xsd/settings-1.0.0.xsd">
  ...
  <servers>
    <server>
      <id>server001</id>
      <username>my_login</username>
      <password>my_password</password>
      <privateKey>${usr.home}/.ssh/id_dsa</privateKey>
      <passphrase>some_passphrase</passphrase>
      <filePermissions>664</filePermissions>
      <directoryPermissions>775</directoryPermissions>
      <configuration></configuration>
    </server>
  </servers>
  ...
</settings>
```

server下的元素包括：

id
: 这是server的id（注意不是用户登陆的id），该id与distributionManagement中repository元素的id相匹配。

username, password
: 这对元素表示服务器认证所需要的登录名和密码。

privateKey, passphrase
: 和前两个元素类似，这一对元素指定了一个私钥的路径（默认是/home/hudson/.ssh/id_dsa）以及如果需要的话，一个密语。将来passphrase和password元素可能会被提取到外部，但目前它们必须在settings.xml文件以纯文本的形式声明。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

filePermissions, directoryPermissions

如果在部署的时候会创建一个仓库文件或者目录，这时候就可以使用权限（permission）。这两个元素合法的值是一个三位数字，其对应了*nix文件系统的权限，如664，或者775。

A.2.3. 镜像（Mirrors）

例 A.4. settings.xml中的Mirror配置

```
<settings xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                      http://maven.apache.org/xsd/settings-1.0.0.xsd">
  ...
  <mirrors>
    <mirror>
      <id>planetmirror.com</id>
      <name>PlanetMirror Australia</name>
      <url>http://downloads.planetmirror.com/pub/maven2</url>
      <mirrorOf>central</mirrorOf>
    </mirror>
  </mirrors>
  ...
</settings>
```

id, name

该镜像的唯一定义符。id用来区分不同的mirror元素。

url

该镜像的URL。构建系统会优先考虑使用该URL，而非使用默认的服务器URL。

mirrorOf

被镜像的服务器的id。例如，如果我们要设置了一个Maven中央仓库（http://repo1.maven.org/maven2）的镜像，就需要将该元素设置成central。这必须和中央仓库的id central完全一致。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

A.2.4. 代理（Proxies）

例 A.5. settings.xml中的proxy配置

```
<settings xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                      http://maven.apache.org/xsd/settings-1.0.0.xsd">
  ...
  <proxies>
    <proxy>
      <id>myproxy</id>
      <active>true</active>
      <protocol>http</protocol>
      <host>proxy.somewhere.com</host>
      <port>8080</port>
      <username>proxyuser</username>
      <password>somepassword</password>
      <nonProxyHosts>*.google.com|ibiblio.org</nonProxyHosts>
    </proxy>
  </proxies>
  ...
</settings>
```

id
: 该代理的唯一定义符，用来区分不同的proxy元素。

active
: `true`则激活代理。当我们声明了一组代理，而某个时候只需要激活一个代理的时候，该元素就可以派上用处。

protocol, host, port
: 该代理的`protocol://host:port`，（协议://主机名:端口），分隔成离散的元素以方便配置。

username, password
: 这一对元素表示代理服务器认证的登录名和密码。

nonProxyHosts
: 这里定义一个不该被代理的主机名列表。该列表的分隔符由代理服务器指定；上述的例子中使用了竖线分隔符，使用逗号分隔也很常见。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

A.2.5. Profiles

settings.xml中的profile元素是pom.xml中profile元素的裁剪版本。它包含了activation, repositories, pluginRepositories 和 properties元素。这里的profile元素只包含这四个子元素是因为这里只关心构建系统这个整体（这正是settings.xml文件的角色定位），而非单独的项目对象模型设置。

如果一个settings中的profile被激活，它的值会覆盖任何其它定义在POM中或者profile.xml中的带有相同id的profile。

A.2.6. 激活（Activation）

Activation是profile的开启钥匙。如POM中的profile一样，profile的力量来自于它能够在某些特定的环境中自动使用某些特定的值；这些环境通过activation元素指定。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

例 A.6. 在settings.xml中定义Activation参数

```
<settings xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                      http://maven.apache.org/xsd/settings-1.0.0.xsd">
  ...
  <profiles>
    <profile>
      <id>test</id>
      <activation>
        <activeByDefault>false</activeByDefault>
        <jdk>1.5</jdk>
        <os>
          <name>Windows XP</name>
          <family>Windows</family>
          <arch>x86</arch>
          <version>5.1.2600</version>
        </os>
        <property>
          <name>mavenVersion</name>
          <value>2.0.3</value>
        </property>
        <file>
          <exists>/usr/local/hudson/hudson-home/jobs/maven-guide-zh-to-production/w
          <missing>/usr/local/hudson/hudson-home/jobs/maven-guide-zh-to-production/
        </file>
      </activation>
      ...
    </profile>
  </profiles>
  ...
</settings>
```

当所有指定的条件符合的时候，profile机会被激活，虽然大部分情况我们不会需要所有的条件。

jdk
: activation通过其jdk元素，提供了一个内置的，Java-核心的检查器。如果我们运行的一个jdk版本号，这里所配置的值能作为前缀与之匹配，激活就会发生。在上述的例子中，1.5.0_06就能够匹配。

os
: os元素可以定义一些操作系统相关的属性，如上例。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

property
: 如果Maven检测到某一个属性（其值可以在POM中通过${名称}引用），其拥有对应的名称和值，Profile就会被激活。

file
: 最后，通过提供一个文件名，通过检测该文件的存在或不存在来激活profile。

activation元素并不是激活profile的唯一方式。settings.xml文件中的activeProfile元素可以包含profile的id。profile也可以通过在命令行，使用-P标记和逗号分隔的列表来显式的激活（如，-P test）。

要了解在某个特定的构建中哪些profile会激活，可以使用maven-help-plugin。

```
mvn help:active-profiles
```

A.2.7. 属性（Properties）

Maven属性和Ant中的属性一样，可以用来存放一些值。这些值可以在POM中的任何地方使用标记${X}来使用，这里X是指属性的名称。属性有五种不同的形式，并且都能在settings.xml文件中访问。

1. env.X: 在一个变量前加上"env."的前缀，会返回一个shell环境变量。例如，/usr/lib/jvm/java-1.6.0-openjdk-1.6.0.0/bin:/usr/local/bin:/usr/local/maven/bin:/usr/kerberos/sbin:/usr/kerberos/bin:/usr/java/latest/bin:/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin:/root/bin:/usr/bin:/usr/local/bin指代了$path环境变量（在Windows上是%PATH%）。

2. project.x：这个点（.）标记的路径，指代了POM中对应的元素值。

3. settings.x: 这个点（.）标记的路径，指代了settings.xml中对应元素的值。

4. Java System Properties: 所有可通过java.lang.System.getProperties()访问的属性都能在POM中使用该形式访问，如/usr/lib/jvm/java-1.6.0-openjdk-1.6.0.0/jre。

5. x: 在<properties/>元素中，或者外部文件中设置，以${someVar}的形式使用。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

例 A.7. 在settings.xml中设置${user.install} 属性

```
<settings xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                      http://maven.apache.org/xsd/settings-1.0.0.xsd">
  ...
  <profiles>
    <profile>
      ...
      <properties>
        <user.install>/ebs1/build-machine/usr/local/hudson/hudson-home/jobs/maven-g
      </properties>
      ...
    </profile>
  </profiles>
  ...
</settings>
```

如果该profile激活，属性`${user.install}`就可以在POM中被访问。

A.2.8. 仓库（Repositories）

仓库是Maven用来填充构建系统本地仓库所使用的一组远程项目。而Maven是从本地仓库中使用其插件和依赖。不同的远程仓库可能含有不同的项目，而在某个激活的profile下，可能定义了一些仓库来搜索需要的发布版或快照版构件。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

例 A.8. settings.xml中的仓库配置

```
<settings xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                      http://maven.apache.org/xsd/settings-1.0.0.xsd">
  ...
  <profiles>
    <profile>
      ...
      <repositories>
        <repository>
          <id>codehausSnapshots</id>
          <name>Codehaus Snapshots</name>
          <releases>
            <enabled>false</enabled>
            <updatePolicy>always</updatePolicy>
            <checksumPolicy>warn</checksumPolicy>
          </releases>
          <snapshots>
            <enabled>true</enabled>
            <updatePolicy>never</updatePolicy>
            <checksumPolicy>fail</checksumPolicy>
          </snapshots>
          <url>http://snapshots.maven.codehaus.org/maven2</url>
          <layout>default</layout>
        </repository>
      </repositories>
      <pluginRepositories>
        ...
      </pluginRepositories>
      ...
    </profile>
  </profiles>
  ...
</settings>
```

releases, snapshots

这里配置了两种构件，发布版（Release）和快照版（Snapshot）的策略。有了这两组配置，POM就可以在每个单独的仓库中，为每种类型类型的构件采取不同的策略。例如，可能有人会决定只为开发的目的开启对快照版本下载的支持。

enabled

`true`或者`false`表示该仓库是否为某种类型构件（发布版或者快照版）开启。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

updatePolicy
该元素指定更新发生的频率。Maven会比较本地POM和远程POM的时间戳。这里的选项是：always（一直），daily（默认，每日），interval：X（这里X是以分钟为单位的时间间隔），或者never（从不）。

checksumPolicy
当Maven将构件部署到仓库中时，它也会部署对应的校验和文件。当没有校验和文件，或者该文件不正确时，你的选项有ignore（忽略），fail（失败），或者warn（警告）。

layout
在上面的仓库描述中，它们都遵循一个共同的布局。大部分情况都是这样。Maven 2为其仓库提供了一个默认的布局；然而，Maven 1.x有一种不同的布局。我们可以使用该元素指定布局是default（默认）还是legacy（遗留）。

A.2.9. 插件仓库

仓库是两种主要构件的家。第一种构件被用作其它构件的依赖。这是中央仓库中存储大部分构件类型。另外一种构件类型是插件。Maven插件是一种特殊类型的构件。由于这个原因，插件仓库独立于其它仓库。pluginRepositories元素的结构和repositories元素的结构类似。每个pluginRepository元素指定一个Maven可以用来寻找新插件的远程地址。

A.2.10. 激活的Profile

例 A.9. 在settings.xml中设置激活profile

```
<settings xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                      http://maven.apache.org/xsd/settings-1.0.0.xsd">
  ...
  <activeProfiles>
    <activeProfile>env-test</activeProfile>
  </activeProfiles>
</settings>
```

settings.xml中最后一需要理解的配置是activeProfiles元素。该元素包含了一组activeProfile元素，每个activeProfile都含有一个profile id。任何在activeProfile中定义的profile id，不论环境设置如何，其对应的profile都会被激活。如果没有匹配的profile，则什么都不会发生。例如，env-test是一个activeProfile，则在pom.xml（或者profile.xml）中对应id的profile会被激活。如果运行过程中找不到这样一个profile，Maven则会像往常一样运行。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

附录 B. 附录：Sun规格说明可选实现

Apache Geronimo项目维护了各种企业级Java规格说明的实现。表 B.1 “规格说明的可选实现构件”列出了所有实现的构件id和版本。要使用这些依赖，使用groupId `org.apache.geronimo.specs`，找到你想要使用的规格说明版本，然后使用表 B.1 “规格说明的可选实现构件”中列出的构件id和构件版本来指向相应的依赖。

注意

所有表 B.1 “规格说明的可选实现构件”中的构件，都拥有同样的groupId：`org.apache.geronimo.specs`。

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

表 B.1. 规格说明的可选实现构件

Corba	2.3	geronimo-corba_2.3_spec	1.1
Corba	3.0	geronimo-corba_3.0_spec	1.2
EJB	2.1	geronimo-ejb_2.1_spec	1.1
EJB	3.0	geronimo-ejb_3.0_spec	1.0
EL	1.0	geronimo-el_1.0_spec	1.0
Interceptor	3.0	geronimo-interceptor_3.0_spec	1.0
J2EE Connector	1.5	geronimo-j2ee-connector_1.5_spec	1.1.1
J2EE Deployment	1.1	geronimo-j2ee-deployment_1.1_spec	1.1
J2EE JACC	1.0	geronimo-j2ee-jacc_1.0_spec	1.1.1
J2EE Management	1.0	geronimo-j2ee-management_1.0_spec	1.1
J2EE Management	1.1	geronimo-j2ee-management_1.1_spec	1.0
J2EE	1.4	geronimo-j2ee_1.4_spec	1.1
JACC	1.1	geronimo-jacc_1.1_spec	1.0
JEE Deployment	1.1MR3	geronimo-javaee-deployment_1.1MR3_spec	1.0
JavaMail	1.3.1	geronimo-javamail_1.3.1_spec	1.3
JavaMail	1.4	geronimo-javamail_1.4_spec	1.2
JAXR	1.0	geronimo-jaxr_1.0_spec	1.1
JAXRPC	1.1	geronimo-jaxrpc_1.1_spec	1.1
JMS	1.1	geronimo-jms_1.1_spec	1.1
JPA	3.0	geronimo-jpa_3.0_spec	1.1
JSP	2.0	geronimo-jsp_2.0_spec	1.1
JSP	2.1	geronimo-jsp_2.1_spec	1.0
JTA	1.0.1B	geronimo-jta_1.0.1B_spec	1.1.1
JTA	1.1	geronimo-jta_1.1_spec	1.1
QName	1.1	geronimo-qname_1.1_spec	1.1
SAAJ	1.1	geronimo-saaj_1.1_spec	1.1
Servlet	2.4	geronimo-servlet_2.4_spec	1.1.1
Servlet	2.5	geronimo-servlet_2.5_spec	1.1.1
STaX API	1.0	geronimo-stax-api_1.0_spec	1.0.1
WS Metadata	2.0	geronimo-ws-metadata_2.0_spec	1.1.1

欢迎加入我们Java总群：309603235，解决各种坑爹的Java问题，群内有Java技术大牛

注意

在你阅读本书的时候，构件版本一列中的版本号可能已经过时了。你可以访问http://repo1.maven.org/maven2/org/apache/geronimo/specs/以检查最新的版本号，点击你想要添加的artifactId。选择你想依赖的规格说明的最高版本号。

这里展示一下如何使用表 B.1 “规格说明的可选实现构件”，如果你项目中有一些代码需要和JTA 1.0.1B规格说明打交道，你就需要往项目中添加如下的依赖：

例 B.1. 添加JTA 1.0.1B到项目中

```
<dependency>
  <groupId>org.apache.geronimo.specs</groupId>
  <artifactId>geronimo-jta_1.0.1B_spec</artifactId>
  <version>1.1.1</version>
</dependency>
```

注意到构件的版本和规格说明的版本并不一致——这里的依赖配置添加了JTA规格说明的1.0.1B版本，但使用的构件版本是1.1.1。在依赖于Geronimo可选实现的时候要意识到这一点，同时一定要确保你使用了该规格说明最新的构件版本号。